

# REPORT DOCUMENTATION PAGE

Form Approved  
OPM No.

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data needed, and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.

1. AGENCY USE (Leave)		2. REPORT		3. REPORT TYPE AND DATES	
4. TITLE AND Electronic Data Systems Corp. Compiler: OC Systems Legacy Ada/370, Release 1.4.1 (without optimization)				5. FUNDING	
6.					
7. PERFORMING ORGANIZATION NAME(S) AND Computer Systems Laboratory (CSL) National Institute of Standards and Technology Building 255, Room A266 Gaithersburg, MD 20899				8. PERFORMING ORGANIZATION	
9. SPONSORING/MONITORING AGENCY NAME(S) AND Ada Joint Program Office Code TXEA, 701 S. Courthouse Rd. Arlington, VA 22204-2199				10. SPONSORING/MONITORING AGENCY	
11. SUPPLEMENTARY					
12a. DISTRIBUTION/AVAILABILITY Approved for Public Release; distribution unlimited				12b. DISTRIBUTION	
13. (Maximum 200 VCL: 941117S1.11380 Host: AMDAHL 5990 (under VM/ESA, Release 2.1)					
14. SUBJECT Ada Programming Language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability Validation Testing, AVO, AVF, ANSI /MIL-STD-1815A, A TPO					
15. NUMBER OF				16. PRICE	
17. SECURITY CLASSIFICATION UNCLASSIFIED		18. SECURITY UNCLASSIFIED		19. SECURITY CLASSIFICATION UNCLASSIFIED	
20. LIMITATION OF UNCLASSIFIED					

DTIC  
ELECTE  
JAN 24 1995  
S G D

19950119 011

DTIC QUALITY INSPECTED 2



AVF Control Number: NIST94EDS504\_1\_1.11  
DATE COMPLETED  
BEFORE ON-SITE:  
AFTER ON-SITE: 94-11-18  
REVISIONS: 94-12-02

Ada COMPILER  
VALIDATION SUMMARY REPORT:  
Certificate Number: 941117S1.11380  
Electronic Data Systems Corp.  
OC Systems Legacy Ada/370, Release 1.4.1 (without optimization)  
AMDAHL 5990 => AMDAHL 5990

Prepared By:  
Software Standards Validation Group  
Computer Systems Laboratory  
National Institute of Standards and Technology  
Building 225, Room A266  
Gaithersburg, Maryland 20899  
U.S.A.

Accession For	
NTIS CRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification .....	
By .....	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	



AVF Control Number: NIST94EDS504\_1\_1.11

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on November 17, 1994.

Compiler Name and Version: OC Systems Legacy Ada/370, Release 1.4.1  
(without optimization)

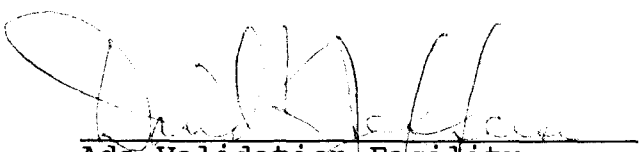
Host Computer System: AMDAHL 5990 under VM/ESA, Release 2.1

Target Computer System: AMDAHL 5990 under VM/ESA, Release 2.1

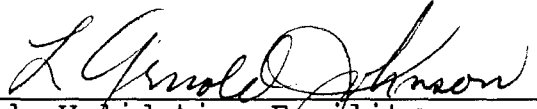
See section 3.1 for any additional information about the testing environment.


As a result of this validation effort, Validation Certificate 941117S1.11380 is awarded to Electronic Data Systems Corp.. This certificate expires 2 years after ANSI/MIL-STD-1815B is approved by ANSI.

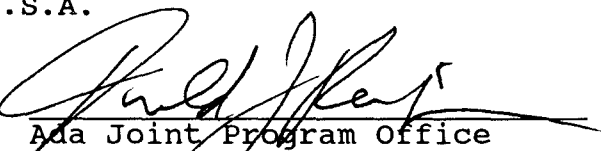
This report has been reviewed and is approved.

  
Ada Validation Facility  
Dr. David K. Jefferson  
Chief, Information Systems  
Engineering Division (ISED)

Computer Systems Laboratory (CSL)  
National Institute of Standards and Technology  
Building 225, Room A266  
Gaithersburg, Maryland 20899  
U.S.A.

  
Ada Validation Facility  
Mr. L. Arnold Johnson  
Manager, Software Standards  
Validation Group

  
Ada Validation Organization  
Director, Computer & Software  
Engineering Division  
Institute for Defense Analyses  
Alexandria VA 22311

  
Ada Joint Program Office  
Donald J. Reifer  
Director, Ada Joint Program Office  
Defense Information Systems Agency,  
Center for Information Management  
Washington DC 20301

U.S.A.



## DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Customer: Electronic Data Systems Corp.  
Certificate Awardee: Electronic Data Systems Corp.  
Ada Validation Facility: National Institute of Standards and  
Technology  
Computer Systems Laboratory (CSL)  
Software Standards Validation Group  
Building 225, Room A266  
Gaithersburg, Maryland 20899  
U.S.A.

ACVC Version: 1.11

## Ada Implementation:

Compiler Name and Version: OC Systems Legacy Ada/370, Release 1.4.1  
(without optimization)  
Host Computer System: AMDAHL 5990 under VM/ESA, Release 2.1  
Target Computer System: AMDAHL 5990 under VM/ESA, Release 2.1

## Declaration:

I the undersigned, declare that I have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.

David C. Pesare  
Customer Signature  
Company Electronic Data Systems Corp.  
Title CONTRACT ADMINISTRATION SPECIALIST

11-16-94  
Date

David C. Pesare  
Certificate Awardee Signature  
Company Electronic Data Systems Corp.  
Title CONTRACT ADMINISTRATION SPECIALIST

11-16-94  
Date



## TABLE OF CONTENTS

### CHAPTER 1 INTRODUCTION

1.1	USE OF THIS VALIDATION SUMMARY REPORT.....	1-1
1.2	REFERENCES.....	1-2
1.3	ACVC TEST CLASSES.....	1-2
1.4	DEFINITION OF TERMS.....	1-3

### CHAPTER 2 IMPLEMENTATION DEPENDENCIES

2.1	WITHDRAWN TESTS.....	2-1
2.2	INAPPLICABLE TESTS.....	2-1
2.3	TEST MODIFICATIONS.....	2-4

### CHAPTER 3 PROCESSING INFORMATION

3.1	TESTING ENVIRONMENT.....	3-1
3.2	SUMMARY OF TEST RESULTS.....	3-1
3.3	TEST EXECUTION.....	3-2

### APPENDIX A MACRO PARAMETERS

### APPENDIX B COMPILATION SYSTEM OPTIONS

### APPENDIX C APPENDIX F OF THE Ada STANDARD



## CHAPTER 1

### INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro92] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro92]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

#### 1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service  
5285 Port Royal Road  
Springfield, Virginia 22161  
U.S.A.

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization  
Computer and Software Engineering Division  
Institute for Defense Analyses  
1801 North Beauregard Street  
Alexandria, Virginia 22311-1772  
U.S.A.



## 1.2 REFERENCES

- [Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
- [Pro92] Ada Compiler Validation Procedures, Version 3.1, Ada Joint Program Office, August 1992.
- [UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

## 1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK\_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK\_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK\_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values--for example, the



largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 3.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

#### 1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, Validation consisting of the test suite, the support programs, the ACVC Capability User's Guide and the template for the validation summary (ACVC) report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification Office system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.



Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.
Conformity	Fulfillment by a product, process, or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable Test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A -1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.



Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro92].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn Test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.



## CHAPTER 2

### IMPLEMENTATION DEPENDENCIES

#### 2.1 WITHDRAWN TESTS

Some tests are withdrawn by the AVO from the ACVC because they do not conform to the Ada Standard. The following 104 tests had been withdrawn by the Ada Validation Organization (AVO) at the time of validation testing. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 93-11-22.

B27005A	E28005C	B28006C	C32203A	C34006D	C35507K
C35507L	C35507N	C35507O	C35507P	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	C37310A	B41308B
C43004A	C45114A	C45346A	C45612A	C45612B	C45612C
C45651A	C46022A	B49008A	B49008B	A54B02A	C55B06A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
B83026B	C83026A	C83041A	B85001L	C86001F	C94021A
C97116A	C98003B	BA2011A	CB7001A	CB7001B	CB7004A
CC1223A	BC1226A	CC1226B	BC3009B	BD1B02B	BD1B06A
AD1B08A	BD2A02A	CD2A21E	CD2A23E	CD2A32A	CD2A41A
CD2A41E	CD2A87A	CD2B15C	BD3006A	BD4008A	CD4022A
CD4022D	CD4024B	CD4024C	CD4024D	CD4031A	CD4051D
CD5111A	CD7004C	ED7005D	CD7005E	AD7006A	CD7006E
AD7201A	AD7201E	CD7204B	AD7206A	BD8002A	BD8004C
CD9005A	CD9005B	CDA201E	CE2107I	CE2117A	CE2117B
CE2119B	CE2205B	CE2405A	CE3111C	CE3116A	CE3118A
CE3411B	CE3412B	CE3607B	CE3607C	CE3607D	CE3812A
CE3814A	CE3902B				

#### 2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. The inapplicability criteria for some tests are explained in documents issued by ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.



The following 201 tests have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

C24113L..V (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

The following 20 tests check for the predefined type `LONG_INTEGER`; for this implementation, there is no such type:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45613C	C45614C	C45631C	C45632C	B52004D
C55B07A	B55B09C	B86001W	C86006C	CD7101F

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than `INTEGER`, `LONG_INTEGER`, or `SHORT_INTEGER`; for this implementation, there is no such type.

C35713B, C45423B, B86001T, and C86006H check for the predefined type `SHORT_FLOAT`; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than `FLOAT`, `LONG_FLOAT`, or `SHORT_FLOAT`; for this implementation, there is no such type.

C45423A, C45523A, and C45622A check that the proper exception is raised if `MACHINE_OVERFLOW` is `TRUE` and the results of various floating-point operations lie outside the range of the base type; for this implementation, `MACHINE_OVERFLOW` is `FALSE`.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a `SYSTEM.MAX_MANTISSA` of 47 or greater; for this implementation, `MAX_MANTISSA` is less than 47.

B86001Y uses the name of a predefined fixed-point type other than type `DURATION`; for this implementation, there is no such type.

CA2009C and CA2009F check whether a generic unit can be instantiated before its body (and any of its subunits) is compiled; this implementation creates a dependence on generic units as allowed by AI-00408 and AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (See section 2.3.)



CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions; this implementation provides no package MACHINE\_CODE.

AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT\_IO with unconstrained array types and record types with discriminants without defaults; these instantiations are rejected by this compiler.

The 21 tests listed in the following table check that USE\_ERROR is raised if the given file operations are not supported for the given combination of mode and access method; this implementation supports these operations.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	-----	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

The following 16 tests check operations on sequential, direct, and text files when multiple internal files are associated with the same external file and one or more are open for writing; USE\_ERROR is raised when this association is attempted.



CE2107B..E	CE2107G..H	CE2107L	CD2110B	CE2110D
CE2111D	CE2111H	CE3111B	CE3111D..E	CE3114B
CE3115A				

CE2203A checks that WRITE raises USE\_ERROR if the capacity of an external sequential file is exceeded; this implementation cannot restrict file capacity.

CE2403A checks that WRITE raises USE\_ERROR if the capacity of an external direct file is exceeded; this implementation cannot restrict file capacity.

CE3413B checks that PAGE raises LAYOUT\_ERROR when the value of the page number exceeds COUNT'LAST; for this implementation, the value of COUNT'LAST is greater than 150000, making the checking of this objective impractical.

LA3004A..B, EA3004C..D, and CA3004E..F (6 tests) check pragma INLINE for procedures and functions; this implementation does not support pragma INLINE.

### 2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 28 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

BA1001A1	BA2001C	BA2001E2	BA3006A6M	BA3006B3
BA3007B7	BA3008A4	BA3008B5	BA3013A6	BA3013A7M

C52008B was graded passed by Test Modification as directed by the AVO. This test uses a record type with discriminants with defaults; this test also has array components whose length depends on the values of some discriminants of type INTEGER. On elaboration of the type declaration, this implementation raises NUMERIC\_ERROR as it attempts to calculate the maximum possible size for objects of the type. The AVO ruled that this behavior was acceptable, and that the test should be modified to constrain the subtype of the discriminants. Line 16 was modified to create a constrained subtype of INTEGER, and discriminant specifications in lines 17 and 25 were modified to use that subtype; these lines are given below:

```

16  SUBTYPE SUBINT IS INTEGER RANGE -128..127;
17  TYPE REC1(D1,D2 : SUBINT) IS

25  TYPE REC2(D1,D2,D3,D4 : SUBINT := 0) IS

```



CA2009C and CA2009F were graded inapplicable by Evaluation Modification as directed by the AVO. These tests contain instantiations of a generic unit prior to the compilation of that unit's body; as allowed by AI-00408 and AI-00506, the compilation of the generic unit bodies makes the compilation unit that contains the instantiations obsolete.

BC3204C and BC3205D were graded passed by Processing Modification as directed by the AVO. These tests check that instantiations of generic units with unconstrained types as generic actual parameters are illegal if the generic bodies contain uses of the types that require a constraint. However, the generic bodies are compiled after the units that contain the instantiations, and this implementation creates a dependence of the instantiating units on the generic units as allowed by AI-00408 and AI-00506 such that the compilation of the generic bodies makes the instantiating units obsolete--no errors are detected. The processing of these tests was modified by re-compiling the obsolete units; all intended errors were then detected by the compiler.

CD1009A, CD1009I, CD1C03A, CD2A21C, CD2A24A, CD2A31A, CD2A31B, and CD2A31C were graded passed by Evaluation Modification as directed by the AVO. These tests use instantiations of the support procedure `LENGTH_CHECK`, which uses `Unchecked_Conversion` according to the interpretation given in AI-00590. The AVO ruled that this interpretation is not binding under ACVC 1.11; the tests are ruled to be passed if they produce Failed messages only from the instances of `LENGTH_CHECK`--i.e., the allowed `Report.Failed` messages have the general form:

" \* CHECK ON REPRESENTATION FOR <TYPE\_ID> FAILED."

EE3301B, EE3405B, and EE3410F were graded passed by Evaluation Modification as directed by the AVO. These tests check certain I/O operations on the current default output file, including standard output. This implementation outputs the ASCII form-feed character which has no effect on the standard IBM output devices; in general, there is no common form-feed mechanism for the devices. Thus, the printed output from this test did not contain the expected page breaks. The AVO ruled that these tests should be considered passed if none of the internal check of the tests were failed (i.e., if the tests report "TENTATIVELY PASSED").



CE2103C..D (2 tests) were graded passed by Evaluation Modification as directed by the AVO. The tests close an empty file; however, the IBM VM/ESA operating system does not allow an empty file to exist, and so the file is deleted and USE\_ERROR is raised. The AVO ruled that this behavior is acceptable, given the operating system (cf. AI-00325); AVO directed that the tests be modified and passed with the following write statement inserted into the two tests, respectively, at lines 56 and 55:

```
WRITE (TEST_FILE_ONE, '1');
```



CHAPTER 3  
PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For sales information about this Ada implementation, contact:

Mr. Charles R. Ware  
Account Manager  
C/O EDS  
13600 EDS Drive  
Mail Stop A3S-B50  
Herndon, VA 22071 (U.S.A.)  
Voice: 703-733-3230  
FAX: 703-733-3240

For technical information about this Ada implementation, contact:

Mr. Christopher K. Anderson  
Technical Manager  
C/O EDS  
13600 EDS Drive  
Mail Stop A3S-B50  
Herndon, VA 22071 (U.S.A.)  
Voice: 703-733-3260  
FAX: 703-733-3240

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro92].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.



The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system--if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

a) Total Number of Applicable Tests	3760	
b) Total Number of Withdrawn Tests	104	
c) Processed Inapplicable Tests	306	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	0	
f) Total Number of Inapplicable Tests	306	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

### 3.3 TEST EXECUTION

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled, linked, and executed on the host/target computer system, as appropriate. The results were captured on the host/target computer system.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

ADA filename {options}

where filename specifies the file to be compiled.

options	description
ERROR(LIST)	Creates a listing file only when errors are encountered. The file contains compile-time error messages



COMPILE | MAIN | BIND

interspersed with the source code. Compile is the default option causing a compile only. BIND will be used in those instances for subunits needing to be compiled prior to the main program. MAIN is specified for mains and will allow execution to take place.

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.



## APPENDIX A

### MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for \$MAX\_IN\_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
-----	
\$MAX_IN_LEN	200 -- Value of V
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
\$BIG_STRING1	'"' & (1..V/2 => 'A') & "'"
\$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & "'"
\$BLANKS	(1..V-20 => ' ')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
\$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & "'"



The following table contains the values for the remaining macro parameters.

Macro Parameter	Macro Value
\$ACC_SIZE	32
\$ALIGNMENT	4
\$COUNT_LAST	2_147_483_646
\$DEFAULT_MEM_SIZE	16777215
\$DEFAULT_STOR_UNIT	8
\$DEFAULT_SYS_NAME	IBM370
\$DELTA_DOC	2#1.0#E-31
\$ENTRY_ADDRESS	ENT_ADDRESS
\$ENTRY_ADDRESS1	ENT_ADDRESS1
\$ENTRY_ADDRESS2	ENT_ADDRESS2
\$FIELD_LAST	1000
\$FILE_TERMINATOR	' '
\$FIXED_NAME	NO_SUCH_FIXED_TYPE
\$FLOAT_NAME	NO_SUCH_TYPE
\$FORM_STRING	""
\$FORM_STRING2	CONNOT_RESTRICT_FILE_CAPACITY
\$GREATER_THAN_DURATION	86401.0
\$GREATER_THAN_DURATION_BASE_LAST	131073.0
\$GREATER_THAN_FLOAT_BASE_LAST	7.237006E+75
\$GREATER_THAN_FLOAT_SAFE_LARGE	7.237004E+75
\$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	



	7.237E+75
\$HIGH_PRIORITY	255
\$ILLEGAL_EXTERNAL_FILE_NAME1	BADCHAR*%
\$ILLEGAL_EXTERNAL_FILE_NAME2	BAD-CHAR!@
\$INAPPROPRIATE_LINE_LENGTH	2000
\$INAPPROPRIATE_PAGE_LENGTH	-1
\$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.TST")
\$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006F1.TST")
\$INTEGER_FIRST	-2147483648
\$INTEGER_LAST	2147483647
\$INTEGER_LAST_PLUS_1	2147483648
\$INTERFACE_LANGUAGE	C
\$LESS_THAN_DURATION	-86401.0
\$LESS_THAN_DURATION_BASE_FIRST	131073.0
\$LINE_TERMINATOR	' '
\$LOW_PRIORITY	0
\$MACHINE_CODE_STATEMENT	NULL;
\$MACHINE_CODE_TYPE	NO_SUCH_TYPE
\$MANTISSA_DOC	31
\$MAX_DIGITS	15
\$MAX_INT	2147483647
\$MAX_INT_PLUS_1	2147483648
\$MIN_INT	-2147483648
\$NAME	NO_SUCH_TYPE_AVAILABLE
\$NAME_LIST	MC68000,ANUYK44,IBM370



\$NAME_SPECIFICATION1	X2102A DATA A1
\$NAME_SPECIFICATION2	X2102B DATA A1
\$NAME_SPECIFICATION3	X3119A DATA A1
\$NEG_BASED_INT	16#FFFFFFFFE#
\$NEW_MEM_SIZE	16777215
\$NEW_STOR_UNIT	8
\$NEW_SYS_NAME	IBM370
\$PAGE_TERMINATOR	' '
\$RECORD_DEFINITION	NEW INTEGER;
\$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
\$TASK_SIZE	32
\$TASK_STORAGE_SIZE	1024
\$TICK	0.000001
\$VARIABLE_ADDRESS	VAR_ADDRESS
\$VARIABLE_ADDRESS1	VAR_ADDRESS1
\$VARIABLE_ADDRESS2	VAR_ADDRESS2
\$YOUR_PRAGMA	PRIORITY



## APPENDIX B

### COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.



## Chapter 2. Compiling Ada Programs

This chapter describes how you can compile Ada/370 programs under the VM/CMS and the MVS TSO or batch environments. It also describes some Ada tools to help you plan your order of compilations. You can refer to special sections at the end of the chapter for additional compilation information on generic units and passive tasks.

If you need help getting started with IBM Ada/370, see Chapter 10, "IBM Ada/370 Tutorial" on page 10-1.

**Note:** If ISPF/PDF is installed under MVS, you can invoke the Ada compiler by using the ISPF/PDF panels described in "ISPF/PDF Panels (MVS Only)" on page 2-28.

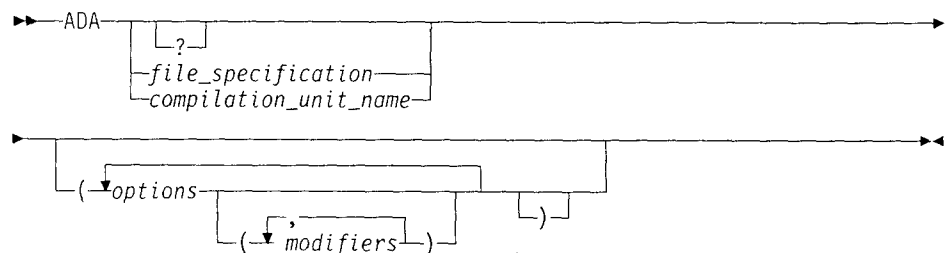
Before reading this chapter, you should be familiar with the creation of source files under your operating system and understand LRM chapter 10.

### Compiling a Source Program

You use the ADA command to compile a source program. The following sections show you how to use this command under VM/CMS and MVS TSO.

See the "Using Ada/370 Compiler and Tools" on page 1-4 for additional guidelines on using the ADA command.

### ADA Command



The "?" option displays syntax information, including a list of the ADA options, on the screen.

Most situations require that you provide the file name of the source file. The compilation unit name is required with certain options. These requirements are discussed in the descriptions of the compiler options.

When you specify a compiler option in ADA, you can use the minimum unique abbreviation. For example, you can specify CReate as CR.

Many compiler options are matched by an opposite option. For example, the opposite of the MAP option is the NOMap option. One of the options is the default, as indicated by an underscore in the option descriptions. The compiler uses the default settings unless you override them by specifying other options to ADA.



Precede the list of options by a blank space and a left parenthesis, and separate them from each other by blank spaces. A closing parenthesis is optional.

Some of the options have modifiers, which you must enclose in parentheses. Where you can enter multiple modifiers to an option, such as with the Xref option, separate the modifiers with a comma.

**Note:** You must specify the sublibraries that your Ada/370 program requires in your library file prior to invoking the ADA command. See "Library Files" on page 5-2 on how you can specify sublibraries in your library file. See the list of "IBM Sublibraries" on page 5-5.

### ***Under VM/CMS:***

To compile Ada programs under VM/CMS, you should have a virtual storage of 10 MB; 9 MB is the minimum. However, you may need more than 10 MB if you are compiling large or complex programs, or when you are compiling a large number of units with input lists.

The *file\_specification* consists of *file\_name*, *file\_type* and *file\_mode*. The *file\_type* and *file\_mode* default to ADA and \*, respectively.

The *compilation\_unit\_name* is the name of the compilation unit.

Here are some examples of the ADA command:

1. ADA EXAMPLE  
compiles EXAMPLE ADA \* using the default options.
2. ADA EXAMPLE (LIBRARY (DEMO LIBRARY) DEBUG  
compiles EXAMPLE ADA \* with the LIBRARY and DEBUG options. LIBRARY causes the compiler to use a file containing an alternative library. The library has the name DEMO LIBRARY \*.
3. ADA EXAMPLE (XREF (BYUNIT,FULL)  
compiles EXAMPLE ADA \* with the Xref option to produce a cross-reference listing. The listing is ordered by compilation unit and includes cross-references to all visible units.

### ***Under MVS TSO:***

Specify the *file\_specification* with a data-set name partially qualified (the default), or fully qualified with single quotation marks.

The *compilation\_unit\_name* is the name of the compilation unit.

Here are some examples of the ADA command:

1. ADA EXAMPLE  
compiles *qualifier*.EXAMPLE using the default options.
2. ADA EXAMPLE (LIBRARY ('DEMO.LIBRARY') DEBUG  
compiles *qualifier*.EXAMPLE with the LIBRARY and DEBUG options. LIBRARY causes the compiler to use a data set containing an alternative library. The library has the name DEMO.LIBRARY. The compiler also saves information needed by the IBM Ada/370 debugger.



## 3. ADA 'USER1.EXAMPLE' (XREF (BYUNIT,FULL))

compiles EXAMPLE with the high-level qualifier USER1. It includes the Xref option to produce a cross-reference listing. The listing is organized by compilation unit and includes cross-references to all visible units.

## The Compiler Options

The ADA command invokes the IBM Ada/370 compiler. Table 2-1 summarizes the compiler options. Square brackets enclose optional modifiers; you do not actually enter the brackets. For the specific syntax of each option, see the option descriptions on the pages specified in the table.

Table 2-1 (Page 1 of 2). Compiler Options			
Option	Default	Function	Page
Asm	NOAsm	Assembler listing.	2-5
[NOGen		Suppress listing of expanded generics.	
NOSys]		Suppress listing of system-supplied generics.	
Bind	COmpile	Bind previously compiled main unit.	2-6
CHeck	COmpile	Compile with syntactic and semantic checking only.	2-6
[Nosemantic]		Compile with syntactic checking only.	
CLear	CLear	Enable automatic clearing of the terminal screen by the compiler.	2-7
NOCLear		Suppress automatic clearing of the terminal screen by the compiler.	
COmpile	COmpile	Compile code for a library unit.	2-7
CRreate [number_of_units]	NOCreate	Initialize working sublibrary for the compiler. <i>number_of_units</i> is number of compilation units in sublibrary.	2-7
DEbug	NODebug	Produce information for debugging.	2-8
DIcompile	COmpile	Compile all deferred instances within the compilation unit.	2-9
ERror	NOERror	Specify action to be taken when errors occur. Must include at least one modifier.	2-9
[Count= <i>number</i> ]		Abort compilation after <i>number</i> errors.	
[List]		Generate interspersed listing of errors and source code.	
EXport	NOEXport	Specify that a non-Ada main program is to be linked with an output object file. Use with MAIn, Bind, or Run options.	2-10
NOEXport			
GEnerate	NOGEnerate (VM/CMS)	Generate a load image.	2-10
NOGEnerate	GEnerate (MVS)		
GRaph	NOGRaph	Produce a call graph listing.	2-11



## The Compiler Options

Table 2-1 (Page 2 of 2). Compiler Options

Option	Default	Function	Page
INList [ <i>max_number</i> ]	CCompile	Compile multiple source files with one invocation of the compiler. <i>max_number</i> is the maximum number of compilation failures during input list processing.	2-11
INSTantiation  [IMmediate  IDeferred  IInline]	INSTantiation [IMmediate]	Compile generic instantiations  Instantiate and compile generic units in instance subunits.  Create the instance subunit, but do not instantiate the generic unit.  Instantiate and compile generic units inline. Pragma SHARE_GENERIC overrides this.	2-11
LIBrary <i>library_name</i>		Specify Ada library name.	2-12
LIST	NOList	Generate interspersed listing of errors and source code.	2-13
MAIn [ <i>compilation_unit_name</i> ]	CCompile	Compile and bind code for a main unit.	2-13
MAP	NOMap	Produce linkage map during binding. Use with MAIn, Bind, or Run options.	2-14
NOCCompile	CCompile	Suppress the compiler.	2-14
Optimize  [Autoinline]	NOOptimize	Optimize the generated object code.  Inline all subprograms that are small or called from one place.	2-15
Passive	NOPassive	Recognize and transform passive tasks.	2-15
Run	NORun	Execute main program.	2-15
SHared	NOSHared	Used with binder, generate a report that describes the sharing of generics in the binder listing file. Use with MAIn, Bind, or Run options.	2-15
Suppress  [Lineinfo]  [Checks I Elab]	NOSuppress	Suppress selected run time checks or line information tables in generated object code. Must include at least one modifier.  Suppress generation of line information tables.  Suppress most run time checks.  Suppress only elaboration checks.	2-16
Trace	NOTrace	Display diagnostic messages from the compiler.	2-16
Usedd( <i>ddname</i> )		Specify the DD names of the data sets that have already been preallocated.	2-16
Xref  [Byunit]  [Full]	NOXref	Produce a cross-reference listing.  Order the listing by compilation unit.  Cross-reference all visible units.	2-17



## Detailed Descriptions of Compiler Options

In the syntax diagrams, uppercase characters indicate the minimum abbreviation of options and their modifiers. Underscored options and modifiers are the defaults.

### Asm



Produces pseudo-assembler language for the object code interspersed with the Ada source for each compilation unit. Asm creates one listing for each source file. This listing, called the data map, also provides information on the relative offset and size allocation of each data item or constant.

For more information on the listing produced by Asm, see "Assembler Listings" on page 8-5.

If you use the Asm option at the same time you invoke the IBM Ada/370 binder, the command also produces a binder listing. This listing contains ADASMAIN, the main program entry point. For more information on the contents of this listing, see chapter 2 of the *Programmer's Guide*.

The NOGen modifier suppresses the listing of code generated for expanded generics. Without NOGen, listings include the code generated for all expanded generics.

The NOSys modifier suppresses the listing of code generated for system-supplied generics.

Use NOGen or NOSys to reduce the size of listings.

#### **Under VM/CMS:**

The name of the listing file takes the form *source* LISTING A, where *source* is the file name of the source file.

The binder listing file name is *source*\$ LISTING A. If *source* is eight characters, the last character is replaced by the \$.

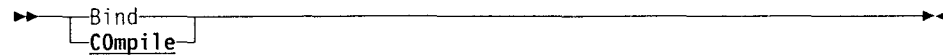
#### **Under MVS:**

The name of the listing data set takes the form *qualifier*.LISTING(*source*) where *qualifier* is the TSO profile prefix and *source* can be either the name of the member of a partitioned data set (PDS) used as source or the second qualifier in the name of a sequential data set. In the case of a binder listing, *source* (as a member name, or as a second level qualifier) has a \$ appended.



## Check

## Bind



Binds a main program that has been previously compiled as a library unit. It produces an object file as output. Enter the compilation unit name in place of the source file name. You can only use the Bind option for compilation units that reside in the working sublibrary of the Ada program library.

To invoke the binder when you compile the source use the MAIn option instead of making a separate call to ADA

You cannot use the Bind option in combination with MAIn, CCheck, DIcompile, C0mpile, or NOC0mpile. If you do, only the last one on the command line is accepted.

### **Under VM/CMS:**

The object file created by compiling with the Bind option has the file name of *comp\_unit* which comes from the name of the compilation unit with the underscores removed and truncated to eight characters. The file type and file mode are TEXT and A, respectively.

### **Under MVS:**

The object file is in a PDS created by compiling with the Bind option. It takes the form *qualifier.OBJ(comp\_unit)*, where *qualifier* is your TSO profile prefix and *comp\_unit* comes from the name of the compilation unit with the underscores removed and truncated to eight characters.

## Check



Causes the compiler to perform only syntactic and semantic error checking. Because no object code is produced, you can save compilation time and disk space during error checking. If you include the Nosemantic modifier, the compiler performs only syntactic error checking.

When you use the CCheck option on a compilation unit with dependents, you must also use the same check option when you compile the dependents. When CCheck(Nosemantic) is specified, the compiler always displays error message EVGDRV3712I and sets the return code to 8 to indicate the library is not updated for CCheck(Nosemantic).

You cannot use the Check option in combination with MAIn, Bind, DIcompile, C0mpile, or NOC0mpile. If you do, only the last one on the command line is accepted.



**Clear**

```

>> Clear
   NOClear

```

Enable automatic clearing of the terminal screen by the compiler before processing begins, and also in between each input list item. NOClear suppresses automatic clearing of the terminal screen.

**Note:** These options are accepted by the compiler, but have no meaning in MVS batch.

**Compile**

```

>> CCompile

```

Causes all compilation units in the source file to be library units, rather than main units. You can make a library unit into a main unit using the Bind option.

You cannot use the CCompile option in combination with MAIn, Bind, Check, DICompile, or NOCCompile. If you do, only the last one on the command line is accepted.

**Create**

```

>> Create
   (number_of_units)
   NOCreate

```

Initializes the working sublibrary for the compiler. The compiler creates a new sublibrary, deleting the previous copy, if one exists.

The *number\_of\_units* variable specifies the number of compilation units the sublibrary can contain. The default is 200; the largest number a sublibrary can contain is 4671.

This number indicates an approximate size for the sublibrary. The number of units that actually fit into a sublibrary depends upon their size and complexity. For further information on sublibraries, see Chapter 5, "Working with the Ada Library System" on page 5-1.

When you use CCreate in conjunction with the LIBrary option, it initializes the working sublibrary in the library specified by LIBrary.

You cannot use CCreate in combination with the Bind option.

**DDnames (MVS JCL Only)**

**Note:** Do not use this option in conjunction with the ADA command. It should only be used in conjunction with the EVGADAB, EVGADAC, EVGADAI, and EVGADABI JCL procedures. Instead, use the Usedd option with the ADA command.



## Debug

►► DDnames — ( — old\_name=new\_name — ) —►►

Specifies the Data Definition (DD) names that identify the data sets used by the compiler and binder. DDnames always requires a value.

For use with the compiler, *old\_name* has one of the following values:

ADAIN  
ADAINFO  
ADALIB  
ADALIST  
ADAUT1  
ADAUT2  
ADAUT3  
ADAUT4

For use with the binder, *old\_name* has one of the following values:

ADAINFO  
ADALIB  
ADALIST  
ADAUT1  
ADAUT2  
ADAUT3  
ADAUT4

Usually, you do not need to change the DD names associated with the compiler because you would use the EVGADAC or EVGADAB procedures. However, you may use your own procedures to invoke the compiler and binder. In this case, your procedure may have DD names different from the ones in EVGADAC or EVGADAB. If so, use this DDnames option to associate your DD names with the data set names used by the compiler and the binder by referring to the DD names in the EVGADAC or EVGADAB procedures.

## Debug

►► DEbug — NODEbug —►►

Causes debugging information used by the IBM Ada/370 debugger to be placed in the working sublibrary. When used with the MAIN or Bind options, DEbug produces a debugging map, which is required by the debugger. You cannot use the DEbug and the Optimize options in the same command.

For more information about debugging, see Chapter 9, The IBM Ada/370 Debugger.

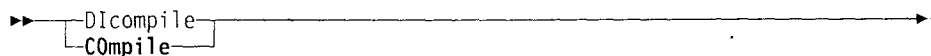
### ***Under VM/CMS:***

The debug map file is created with the name *comp\_unit* DEBUGMAP A, where *comp\_unit* comes from the name of the compilation unit with the underscores removed and truncated to eight characters.



**Under MVS:**

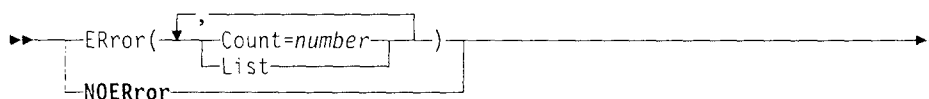
The debug map file is created with the name *qualifier.DEBUGMAP(comp\_unit)*, where *qualifier* is the high-level qualifier and *comp\_unit* comes from the name of the compilation unit with the underscores removed and truncated to eight characters.

**Dicompile**

Compiles all deferred generic instances within the specified compilation unit. When the instantiations of generic units have been deferred by using *INSTantiation* (Deferred), use this *Dicompile* option to compile the instance subunits before binding. See "Instantiation" on page 2-11 for more information.

You also use the *Dicompile* option on compilation units which have instance subunits which are out-of-date. This action only compiles those instance subunits that have not been compiled or are out-of-date.

When you use this option with the ADA command, enter the compilation unit name (not the source file name) that has the instance of the generic unit. You can only use this option for compilation units that reside in the working sublibrary of the Ada program library.

**Error**

Controls the way the compiler behaves when it finds errors in the source file. You must choose at least one of the modifiers.

The *Count* modifier specifies the *number* of errors that cause the compiler to stop processing. The compiler includes syntax and semantic errors in the count. For example, *Count=5* causes the compiler to stop processing after it finds five errors. If you omit the *Count* modifier, the compiler stops processing when it finds 32767 errors, the default error limit. If you give anything except a positive count, error message EVGEXE3032E is issued.

The *LIST* modifier creates a file containing compile-time error messages interspersed with the source code. If there are no errors, the compiler does not generate the listing. To generate this listing regardless, use the *LIST* option describes on 2-13.

With *NOError*, the compiler does not modify its behavior when it finds errors during processing.

**Under VM/CMS:**

The listing file created by the *LIST* modifier is called *source LISTING A*, where *source* is the file name of the source file.

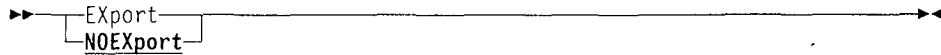


## Generate

### ***Under MVS:***

The listing file created by the LIST modifier is called *qualifier*.LISTING(*source*), where *qualifier* is the TSO profile prefix and *source* is the name of the source data set.

## Export



Indicates that the object file produced by binding a main unit is to be linked with a non-Ada main program that will call an exported Ada subprogram. You can only use the EXport option in conjunction with the Bind option. Refer to page 2-6 for a description of the Bind option.

**Note:** No Ada main program is created when the EXport option is specified.

If the EXport option is specified and the resulting object module does not contain exported code, error message EVGDRV3733W is issued.

## Generate

### ***Under VM/CMS***



### ***Under MVS***



Generates a load image.

When you compile with the GGenerate option, the ADA command also invokes the binder, producing an object file. The object file is then used to produce a load module. This option assumes that the source file contains a main program. It has no effect when you compile the program as a library unit instead of a main unit.

The NOGGenerate option suppresses the invocation of the linkage editor after the main program is bound. Use this option when you want to invoke the linkage editor with options that differ from the default. For example, you would use NOGGenerate when you want to link the main program with non-Ada object code at a later time. You cannot use NOGGenerate in combination with the Run option.

### ***Under VM/CMS:***

The load module created by the GGenerate option has the name *comp\_unit* MODULE A, where *comp\_unit* comes from the name of the compilation unit with the underscores removed and truncated to eight characters.

***Under MVS:*** The load module created by the GGenerate option has the name *qualifier*.LOAD(*comp\_unit*), where *comp\_unit* comes from the name of the compilation unit with the underscores removed and truncated to eight characters.



## Graph



Creates a file containing a call graph listing for each compilation unit contained in the source file. The call graph listing describes the *static* calling relationships between subprograms in the optimized code. For each subprogram in each optimized unit, the call graph lists all possible callers of that subprogram. Call graph listings are useful in analyzing a program to be sure that subprogram calls are necessary.

The GGraph option is effective only when the Optimize option is specified.

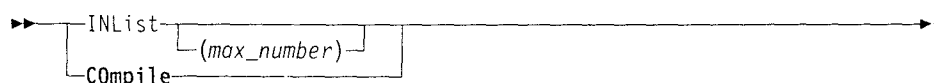
### **Under VM/CMS:**

The compilation listing file is created with the name *source* LISTING A, where *source* is the file name of the source file.

### **Under MVS:**

The compilation listing file is created with the name *qualifier*.LISTING(*source*) where *qualifier* is the TSO profile prefix and *source* can be either the name of the member of a PDS used as the source or the second *qualifier* in the name of a sequential data set.

## Inlist

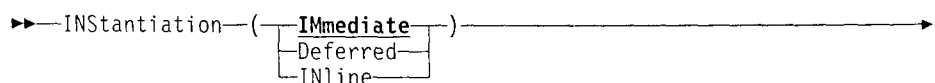


Compiles multiple source files with a single invocation of the compiler. Enter the name of the file containing the input list in place of the source file name. For more information on the use of input lists, see "Compiling Multiple Source Files" on page 2-22.

**Note:** Sequence numbers are not allowed in the input list file and the Ada source file.

If a source file fails to compile, the compiler continues to process the remaining files. You can specify that the compiler stop processing after a certain number of source files fail to compile. To do so, use the *max\_number* variable. If *max\_number* is specified, it must be a positive number, or error message EVGEXE3032E is produced. The default is to continue processing until all compilations are done.

## Instantiation



Specifies whether the compilation of the generic instantiations in the source file is to be done, deferred or inlined. If the INstantiation option is not used then the default is INstantiation (IMmediate).



## Library

The `IMmediate` modifier compiles a generic instantiation similar to an Ada body stub with the generic instance treated as an Ada subunit, referred to as the instance subunit. The body of the generic template must have been compiled. If not, the instantiation method will become deferred and a warning message will be issued.

If you modify a generic body, you only have to recompile the instance subunit by using the `Dlcompile` option with the name of the compilation unit that has the generic instantiation.

The `Deferred` modifier compiles a generic instantiation similar to an Ada body stub with the generic instance treated as an Ada subunit. You use this modifier to defer the instantiation and compilation of the instance subunit (it does not matter if its generic body has been compiled or not). Use the `Dlcompile` option on the compilation unit that has the deferred instance to complete the compilation of the instance subunit.

The `Inline` modifier generates code for the generic instantiation, but it is not treated as a subunit. If the generic template associated with an instance is recompiled, the compilation unit containing the instantiation must be recompiled.

The body of the generic template must be current, available, and must not be optional. Otherwise, the instantiation method will become deferred, and a warning message will be issued.

If you have an error in the generic template, the location of the error message on the console log and in the listing file may change depending on the `INSTantiation` modifier.

## Library

►—LIBRARY—(*library\_name*)—◄

Specifies the name of the Ada library file to be used by the compiler. The *library\_name* modifier is the name of a library file that contains the names of one or more sublibraries. See the list of "IBM Sublibraries" on page 5-5 that you can specify in your library file. You must always include the system sublibrary in your Ada library file.

When you do not specify the `LIBRARY` option, the compiler uses the default library file. Under VM/CMS, it has the name `ADA LIBRARY *`. Under MVS, it has the name *qualifier*.`ADA.LIBRARY`, here *qualifier* is the TSO profile prefix. For information concerning libraries and sublibraries, see Chapter 5, Working with the Ada Library System.

### **Under VM/CMS:**

You can provide the *library\_name* variable in either of two formats. The preferred is *file\_name file\_type file\_mode*. The other format is *file\_mode:file\_name.file\_type*. In both formats, if you specify only the file name, *file\_type* defaults to `LIBRARY` and *file mode* defaults to `"*"`. If you do not specify a library file, ADA searches for `ADA LIBRARY *`.



For example, to specify library PROJ1 LIBRARY A, when you compile the file MYPROG, enter:

```
ADA MYPROG (LIB(PROJ1))
```

Do not select an alternative file type for the Ada library file. Retaining the default file type maintains consistent file naming conventions for all users.

#### ***Under MVS:***

A library can be either a sequential data set or a member of a PDS. The *library\_name* variable can be any valid data-set name format. For example, to specify library PROJ1.LIBRARY on USER1 when you compile the data set MYPROG.SOURCE, enter:

```
ADA 'USER1.MYPROG.SOURCE' (LIB('USER1.PROJ1.LIBRARY'))
```

## List



Creates a file containing a listing for each source file. The listing contains compile-time messages interspersed with the source code. If there are multiple compilation units in a source file, LIST places the listings for all units into a single file.

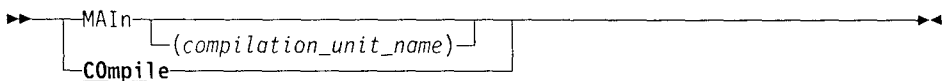
#### ***Under VM/CMS:***

The compilation listing file takes the form *source* LISTING A, where *source* is the file name of the source file.

#### ***Under MVS:***

The compilation listing file takes the form *qualifier*.LISTING(*source*), where *qualifier* is the TSO profile prefix and *source* can be either the name of the member of a PDS used as the source or the second qualifier in the name of a sequential data set.

## Main



Causes the compiler to produce code for the source file as an Ada main program. The MAIn option compiles a program and performs the binding operation without the need to specify any other option.

If the Ada source file contains one or more library compilation units in addition to the main compilation unit, enter the name of the main program in the *compilation\_unit\_name* variable. If you do not specify a unit name, the binder uses the first unit that can be a main subprogram in the file.

You cannot use the MAIn option in combination with Bind, Check, DCompile, COmpile, or NOCOmpile. If you do, only the last one on the command line is accepted.



## Under VM/CMS:

The object file created by compiling with the MAIn option takes the form *comp\_unit* TEXT A, where *comp\_unit* comes from the name of the compilation unit with the underscores removed and truncated to eight characters.

## Under MVS:

The object file created by compiling with the MAIn option takes the form *qualifier*.OBJ(*comp\_unit*), where *qualifier* is the TSO profile prefix and *comp\_unit* comes from the name of the compilation unit with the underscores removed and truncated to eight characters.

# Map



Causes the compiler to produce a linkage map when the IBM Ada/370 binder processes a main program. This file is required by the Ada/370 profiler.

Use MAP in combination with either MAIn or Bind, both of which invoke the binder. You can also use MAP in combination with Run as long as you do not use the NOCOmpile option.

With NOMap, the compiler does not create a linkage map when the IBM Ada/370 binder processes a main program.

## Under VM/CMS:

The map file is created as *comp\_unit* ADAMAP A, where *comp\_unit* is the first eight characters of the compilation unit's name with the underscores deleted.

## Under MVS:

The map file is created as *qualifier*.ADAMAP(*comp\_unit*), where *qualifier* is the TSO profile prefix and *comp\_unit* is the first eight characters of the compilation unit's name with the underscores deleted.

# Nocompile



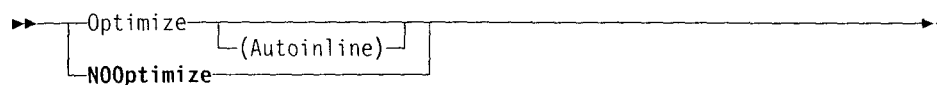
Causes the ADA command to suppress the compilation step. Use NOCOmpile with Run to run an Ada program that has already been compiled. When you use NOCOmpile with Run, enter the compilation unit name in place of the source file name.

You can also use NOCOmpile with the CReate option to create a new working sublibrary without having to compile any source code.

You cannot use the NOCOmpile option in combination with MAIn, Bind, DIcompile, Check, or COmpile. If you do, only the last one on the command line is accepted.



## Optimize



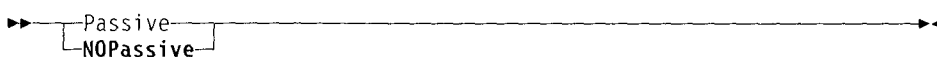
Instructs the compiler to optimize the generated object code. Use `Optimize` to optimize all units compiled.

The `Autoinline` modifier instructs the compiler to inline all small subprograms or subprograms that are called from only one place.

For more information on optimization, see the *Programmer's Guide*.

**Note:** The Ada/370 debugger cannot operate on optimized code.

## Passive



Recognizes tasks that can become passive tasks and transforms them. Passive tasking will only take place when you specify the `Passive` option. See the *Ada/370 Programmer's Guide* for details regarding passive tasking. When you specify the `Optimize` option and the `Passive` option, the compiler will transform passive tasks it recognizes and optimize the code generated for the compilation unit.

## Run



Loads and executes a main program. The compiler assumes that the program is a main unit. You can either compile and run a program, or run a pre-compiled program. To run a previously compiled, bound and link-edited program, use `Run` in combination with the `NOCmpile` option. When you use `Run` with `NOCmpile`, you must specify the compilation unit name, rather than the Ada source file name.

With `NORun`, the compiler does not execute the program.

## Shared

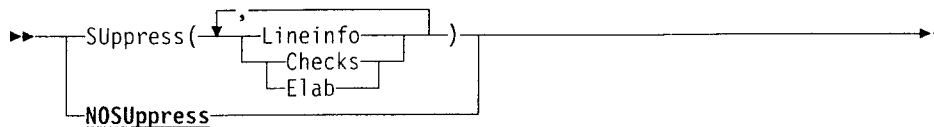


Whenever the binder is invoked with the `Shared` option, a report is generated in the binder listing file to describe the sharing caused by the use of pragma `SHARE_GENERIC`. With `NOShared`, the report is not generated. See "Output for Shared Generic Units" on page 8-19 for a sample of this report.

See *Ada/370 Programmer's Guide* on the use of pragma `SHARE_GENERIC`.



## Suppress



Suppresses selected run time checks and line information in generated object code, resulting in smaller, faster modules. You must choose at least one of the modifiers. Use of the `SUppress` option, `pragma SUPPRESS`, or `pragma SUPPRESS_ALL` causes the compiler to suppress run time checks. You can use `pragma NO_SUPPRESS` to override check suppression within specific compilation units. For more information on these pragmas, see the chapter on tuning in the *IBM Ada/370 Programmer's Guide*.

The `Lineinfo` modifier suppresses the generation of line information tables, saving the space required to produce them. These tables display the Ada source line number when an unhandled exception occurs. If you compile your code with this option and an unhandled exception occurs during run time, the error information does not include a line number.

The `Checks` modifier suppresses most run time checks. See the chapter on tuning in the *IBM Ada/370 Programmer's Guide*.

The `Elab` modifier only suppresses elaboration checks made by other units on this unit. This differs from the way `pragma SUPPRESS` works. It suppresses elaboration checks made on other units from the unit in which it resides.

If you choose both the `Checks` and `Elab` modifiers, the `Checks` modifier takes precedence.

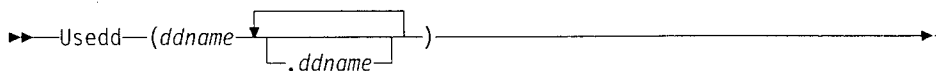
With `NOSuppress`, the compiler does not suppress selected run time checks and line information in generated object code.

## Trace



Displays on the screen the diagnostic messages from the compiler. This option is intended for use in submitting problems to IBM; see the *IBM Ada/370 Diagnosis Guide and Reference*.

## Usedd (MVS Only)



Specifies the DD names of the data sets that you have preallocated to the standard DD names for the ADA command. These data sets are normally allocated by the ADA command.



The ADAIN and ADALIB option modifiers are the standard ADA command DD names of the data sets that you can preallocate. ADAIN is a source or input list data set, and ADALIB is the Ada library data set.

If you specify the data set name or partitioned data set (PDS) member-name parameter, and the ADAIN modifier, the Usedd option modifier takes precedence. Specifying the `LIBRARY(library_name)` option and `Usedd(ADALIB)` causes the `Usedd` option to take precedence.

The following commands provide an example of the `Usedd` option:

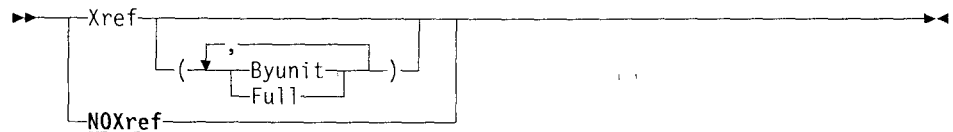
```
ALLOC fi(ADAIN) ds(dsname) shr reu
ADA (Usedd(ADAIN)) MAIN
```

If you do not use the `Usedd` option, you would specify the following:

```
ADA dsname (MAIN
```

**Note:** Do not use `Usedd(ADAIN)` if the file name you specify is a unit name, for example, when using the `Bind` option without the `INList` option. This is only valid when the binder input list facility is not used.

## Xref



Produces a cross-reference listing for each compilation unit contained in the source file. It creates one listing file for each source file.

`Byunit` causes `Xref` to display symbols by compilation unit. By default, the `Xref` listing displays symbols in alphabetic order.

`Full` causes `Xref` to cross-reference each compilation unit with all unit specifications that are visible to it. A unit specification is visible if it is an import to the compilation unit. If the compilation unit is a body, its parent and its parent's imports are also visible. `Full` does not display cross-references for the private parts of imported units. By default, `Xref` only cross-references the compilation units contained in the source file. For more information on listings, see "Cross-Referencer" on page 8-9.

### VM/CMS Usage:

The listing file is created with the name `source LISTING A`, where `source` is the file name of the source file.

### MVS Usage:

The listing file is created with the name `qualifier.LISTING(source)`, where `qualifier` is the TSO profile prefix and `source` can be either the name of the member of a PDS used as source or the second qualifier in the name of a sequential data set.



## Compiling a Program with Job Control Language (JCL)

This section describes how to invoke the compiler as a batch job under MVS using Job Control Language (JCL). A 6MB region is recommended for compiling Ada programs in batch. For information on how to invoke the binder using JCL, see "Using Job Control Language to Bind a Main Program (MVS only)" on page 3-2.

### EVGADAC Cataloged Procedure

The EVGADAC cataloged procedure invokes the compiler to compile Ada programs in a source file or to compile deferred generic instantiations within a compilation unit.

```
//MYPROG    JOB , ' ',MSGCLASS=D,MSGLEVEL=(1,1),NOTIFY=USER1,  
//          CLASS=A,REGION=6000K  
//*  
//*  PURPOSE:  TO RUN THE ADA COMPILER  
//*  
//COMPILE  EXEC PROC=EVGADAC,ADASRC='USER1.ADA.SOURCE(HELLO)',  
//          USER=USER1,CMPPRM='CHECK'
```

*Figure 2-1. Using the EVGADAC Cataloged Procedure to Invoke the Compiler*

The preceding example job, MYPROG, compiles member HELLO in the source PDS, USER1.ADA.SOURCE. The user's name, USER1, is identified with the USER variable. This variable is used as a high-level qualifier to construct data-set names for the compiler, such as USER1.ADA.LIBRARY, which is the default library. Your job card will probably be different, depending on your site's conventions.

After MVS executes this procedure, the Ada program contained in member HELLO is compiled into the working sublibrary of USER1.ADA.LIBRARY.

For more examples of using EVGADAC, see "Examples of Using EVGADAC:" on page 2-22.



A sample EVGADAC cataloged procedure appears in the following figure. The exact location of EVGADAC will depend on your site's conventions, but is installed by default in ADA.V1R4M0.SEVGPRO1.

```
//EVGADAC      PROC CMPPRM=' ',MEMSIZE=8196K,
//            STPLIB='ADA.V1R4M0.SEVGMOD1',MAXTIME=60,
//            ADASRC='NULLFILE', UNIT='',
//            VIO=VIO,SYSDA=SYSALLDA,SYSDA='*'
//*
//*  ERASE ADA.INFO DATASET
//*
//            EXEC PGM=IEFBR14
//ADAINFO      DD DSN=&USER..ADA.INFO,DISP=(MOD,DELETE),
//              SPACE=(1,1),UNIT=&SYSDA
//*****
//*                               INVOKE THE COMPILER
//*****
//STEP1        EXEC PGM=EVGCOMP,PARM='&UNIT (&CMPPRM',REGION=&MEMSIZE,
//              TIME=&MAXTIME,DYNAMNBR=65
//STEPLIB       DD DSN=&STPLIB,DISP=SHR
//CONOUT        DD SYSDA=&SYSDA,DCB=(LRECL=120,BLKSIZE=120)
//ADAIN         DD DSN=&ADASRC,DISP=SHR,FREE=END,DCB=BUFNO=4
//ADALIB        DD DSN=&USER..ADA.LIBRARY,DISP=SHR
//ADAINFO       DD DSN=&USER..ADA.INFO,DISP=(NEW,PASS,CATLG),
//              DCB=(RECFM=VB,LRECL=512,BLKSIZE=3120,DSORG=PS),
//              SPACE=(80,(10,50)),UNIT=&SYSDA
//ADALIST       DD DSN=&USER..LISTING,DISP=(MOD,CATLG,CATLG),
//              DCB=(RECFM=VBA,LRECL=259,BLKSIZE=3120,DSORG=PO,BUFNO=2),
//              SPACE=(132,(500,2000,20)),UNIT=&SYSDA
//ADAUT1        DD SPACE=(22528,(4,10)),
//              DCB=(RECFM=FB,BLKSIZE=4096,DSORG=DA,BUFNO=3),
//              UNIT=&SYSDA
//ADAUT2        DD SPACE=(132,(500,2000)),
//              DCB=(RECFM=VB,LRECL=136,BLKSIZE=3120,DSORG=PS,BUFNO=3),
//              UNIT=&VIO
//ADAUT3        DD SPACE=(22528,(4,10)),
//              DCB=(RECFM=FB,BLKSIZE=4096,DSORG=DA,BUFNO=3),
//              UNIT=&SYSDA
//ADAUT4        DD SPACE=(22528,(4,10)),
//              DCB=(RECFM=FB,BLKSIZE=4096,DSORG=DA,BUFNO=3),
//              UNIT=&SYSDA
```

Figure 2-2. EVGADAC Cataloged Procedure



## Symbolic Variables for the EVGADAC Cataloged Procedure

The EVGADAC cataloged procedure includes several symbolic JCL substitution variables you can modify to specify the various options available.

### Symbolic

#### Variable Description

ADASRC	Specifies the data set name (DSN) for the Ada source file. This has a default of NULLFILE and, if the ADASRC parameter is not used, no JCL error will be issued.
CMPPRM	Specifies options to the compiler in the PARM field. This variable, found in Step 1 of the EVGADAC cataloged procedure, specifies options to the compiler in the PARM field. These options are the same ones used when you invoke the compiler with the ADA command.

The compiler options have the following syntax:

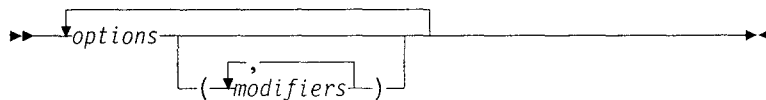


Table 2-2 on page 2-21 lists the valid options, their modifiers, and defaults.

MAXTIME	Sets a maximum amount of time for the compiler job step to run (via the TIME parameter on a JCL EXEC card). The default value is 60 minutes, but this value can be changed when the cataloged procedure is installed.
MEMSIZE	Specifies the amount of memory in which to run the compiler job step (via the REGION option on a JCL EXEC card). The greater the memory, the better the performance of the compiler. The default value is 8196 KB, but this value may be changed, depending on your site's conventions.
STPLIB	Indicates the data set name of the partitioned data set that contains the compiler module, EVGCOMP. This default can be changed when the cataloged procedure is installed.
SYSDA	Specifies the UNIT for permanent data set allocations. The default is SYSALLDA, but this value can be changed depending on your site's conventions.
SYSOUT	Identifies the output class for the compiler output. The default is "*", but this can be changed, depending on your site's conventions.
UNIT	Identifies the compilation unit name. The default is '. '. You would only use it with the DICOPILE compiler option.
USER	Indicates the high-level qualifier that the compiler uses to build data set names. You must specify this variable. It is common to set this variable to your TSO profile prefix.
VIO	Specifies the UNIT for temporary data-set allocations. The default is VIO, but the default can be changed, depending on your site's conventions.



## Compiler Options for the EVGADAC Cataloged Procedure

This section defines the standard options for the CMPPRM symbolic variable when you invoke the compiler as a batch job under MVS. Descriptions of the options appear on the pages shown.

Table 2-2 (Page 1 of 2). Compiler Options for JCL

Option	Default	Function	Page
Asm  [NOGen  I NOSys]	NOAsm	Generate Assembler listing.  Suppress listing of expanded generics.  Suppress listing of system-supplied generics.	2-5
CHeck  [NOSemantic]	COmpile	Compile with syntactic and semantic checking only.  Compile with syntactic checking only.	2-6
CLear  I NOCLear	CLear	Enable automatic clearing of the terminal screen.  Disable automatic clearing of the terminal screen.	2-7
COmpile	COmpile	Compile code for a library unit.	2-7
CReate [number_of_units]	NOCReate	Initialize working sublibrary for the compiler. <i>number_of_units</i> is number of compilation units in sublibrary.	2-7
DDnames old_name=new_name		Specify the Data Definition (DD) names that identify the data sets used by the compiler and binder. It must include at least one modifier.	2-7
DEbug	NODebug	Output information for debugging.	2-8
DICompile	COmpile	Compile all deferred instances within the compilation unit.	2-9
Error  [COUnt=number]  [List]	NOError	Specify action to be taken when errors occur. Must include at least one modifier.  Abort compilation after <i>number</i> errors.  Generate interspersed listing of errors and source code.	2-9
Graph	NOGraph	Produce a call graph listing.	2-11
INSTantiation  [Immediate  IDEFerred  IINLine]	INSTantiation [Immediate]	Compile generic instantiations  Instantiate and compile generic units in instance subunits.  Create the instance subunit, but do not instantiate the generic unit.  Instantiate and compile generic units inline. Pragma SHARE_GENERIC overrides this.	2-11
List	NOList	Generate interspersed listing of errors and source code.	2-13



## Compiling Multiple Source Files

Table 2-2 (Page 2 of 2). Compiler Options for JCL			
Option	Default	Function	Page
Optimize	NOOptimize	Optimize the generated code.	2-15
[AUtoinline]		Inline all subprograms that are small or called from one place.	
Passive	NOPassive	Recognize and transform passive tasks.	2-15
Suppress	NOSuppress	Suppress selected run time checks or line information tables in generated object code. Must include at least one modifier.	2-16
[LIInfo]		Suppress generation of line information tables.	
[CHecks]		Suppress all run time checks.	
[Elab]		Suppress only elaboration checks.	
Trace	NOTrace	Display diagnostic messages from the compiler.	2-16
Xref	NOXref	Produce a cross-reference listing.	2-17
[Byunit]		Order the listing by compilation unit.	
[Full]		Cross-reference all visible units.	

### Examples of Using EVGADAC:

**Note:** The following examples assume that EVGADAC is visible to the job step.

1. `//COMPILE EXEC PROC=EVGADAC,USER=USER1,CMPPRM='NOCOMPILE CREATE'`  
A sublibrary is created.
2. `//COMPILE EXEC PROC=EVGADAC,USER=USER1,CMPPRM='COMPILE'`  
The compiler should open and compile a source file but none was specified using the ADASRC parameter; therefore the following error is generated:  
`EVGAFF5001E >>> Could not open source file =ADAIN`
3. `//COMPILE EXEC PROC=EVGADAC,USER=USER1,CMPPRM='DICOPILE',`  
`// UNIT='X_Y_Z',ADASRC='USER1.ADA.SOURCE(HELLO)`  
The compiler ignores the source file and compiles the deferred instance subunits in the compilation unit X\_Y\_Z.

## Compiling Multiple Source Files

To compile multiple source files in one invocation, you first create an input list. An input list is a file containing a list of the names of files or compilation units to be compiled. See "Creating the Input List" on page 2-23 for the syntax of each line of the input list. You can then compile this input list by using the ADA (INList command on either VM/CMS or MVS, or use the EVGADAI cataloged procedure in MVS JCL batch. See "EVGADAI Cataloged Procedure" on page 2-27.

The names of source files or compilation units appear in the input list, along with other information that controls the compilation process. The compiler processes items in the input list in sequential order. Along with the object code that is the usual result of compilation, the compiler produces a file that contains information on the results of each successful or failed compilation. If you use a compiler option



that produces compilation listings (Asm, LIST, Error, or Xref), the compiler produces a separate listing for each compilation unit.

If the compiler detects errors during compilation of any source file in the list, it goes on to compile the next source file. There may be cases, especially with a large input list, where it is not advisable to continue through the entire input list when multiple source files fail to compile. The `INList` option has a variable that allows you to specify the maximum number of source file compilation failures before the compilation stops.

***Under VM/CMS:***

For example, the command string

ADA MYLIST (INL(6))

compiles the source files in the input list MYLIST INLIST \*, setting the failure limit at six.

**Under MVS:**

For example, the command string

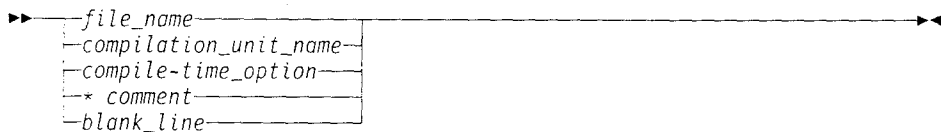
ADA MYLIST.INLIST (INL(6))

compiles the source files in the input list *qualifier*.MYLIST.INLIST, setting the failure limit at six. The compiler uses the default high-level qualifier.

## Creating the Input List

An input list contains two types of information for compilation, names of source files or compilation units and compile-time options.

The syntax for each line of the input list is:



The rules for creating the input list are:

- Place only one file name, one compilation unit name, or one compile-time option on a line.
- Do not place comments on the same line with other information.
- If the first nonblank character in a line is \*, that line is a comment line.
- The compiler ignores blank lines.
- File names or compilation unit names, do not have to start in the first column.



## Compiling Multiple Source Files

This VM/CMS file list follows the rules correctly:

```
* YES, THIS IS A COMMENT
```

```
AFILE ADA A
```

```
BFILE ADA A
```

```
    BISUB ADA A
```

```
    B2SUB ADA A
```

```
CFILE ADA A
```

```
&OPTIONS DICOMPILE
```

```
    UNIT_NAME
```

```
&OPTIONS COMPILE
```

If an error occurs during input list processing, the compiler updates the working sublibrary with information about the units that have been compiled successfully. The compiler also places information about the results into an output file. For more information about the contents of this file, see "Getting Information about an Input List Compilation" on page 2-26.

**Compilation Units:** Compilation unit names in the input list specify the compilation of instance subunits that have been previously deferred or are out-of-date. In this case, the &OPTIONS DICOMPILE must precede the compilation unit name. DICOMPILE remains in effect until you change it, for example with &OPTIONS COMPILE. See "Embedding Compile-Time Options in the Input List" on page 2-25 for more information on how to provide options in the input list.

### VM/CMS Source File Names:

Enter the names of source files into the input list. If you leave out the *file\_type* and *file\_mode*, the compiler assumes that they are ADA \*.

Input List	Compiler Interpretation
------------	-------------------------

MYFILE	MYFILE ADA *
MYFILE TEST	MYFILE TEST *
MYFILE TEST A	MYFILE TEST A

### MVS Source Data Set Names:

Enter the names of source data sets into the input list either fully or partially qualified. Enclose fully qualified names inside a pair of single quotation marks. If you leave out the high-level qualifier, the compiler assumes the current TSO profile prefix.

Input List	Compiler Interpretation
------------	-------------------------

MYFILE	'qualifier.MYFILE'
MYFILE.TEST	'qualifier.MYFILE.TEST'
'OTHER.MYFILE.TEST'	'OTHER.MYFILE.TEST'



### Embedding Compile-Time Options in the Input List

The `&OPTIONS` line is used to place compile-time options and input list options within the input list. It should precede the source file name or compilation unit name that it applies to. The options that are set when you invoke the compiler apply to each source file or compilation unit until a conflicting option embedded in the input list overrides it. Options that appear in the input list apply to all following source files or compilation units until they are overridden by other embedded options. A specific option can appear multiple times in an input list. The options can have the same modifiers as when you specify them on the command line:



The *compiler\_options* you can embed in input lists are:

- Asm
- CHeck
- CLear
- COmpile
- DEBug
- DIcompile
- Error
- Graph
- INList
- INstantiation
- LISt
- NOAsm
- NOClear
- NOCOMpile
- NODEbug
- NOError
- NOGraph
- NOList
- NOOptimize
- NOPassive
- NOSuppress
- NOXref
- Optimize
- Passive
- SUPpress
- Xref

**Note:** INList in an `&OPTIONS` line is only used to change the maximum number of compilation failures allowed during input list processing. It must take a modifier *max\_number*.

There is one valid *input\_list\_option*, `DEFAult`. It causes the compiler to reset all options to their states as set by the ADA command.

The following example shows an input list with embedded options, along with descriptions of how the options change. This example uses VM/CMS file naming conventions; MVS users should use MVS conventions.



### Input List

### Options Used for Compilation

AFILE ADA A	Command-line options
&OPTIONS DEBUG	
BFILE ADA A	Command-line options and DEbug
&OPTIONS NODEBUG	
CFILE ADA A	Command-line options and NODebug
&OPTIONS DEFAULT	
EFILE ADA A	Command-line options only
&OPTIONS INSTANTIATION(DEFERRED)	
FFILE ADA A	Command-line options plus defer generic instantiations
&OPTIONS DICOMPILE	
HELLO_WORLD	Compile instance subunits in HELLO_WORLD compilation unit
&OPTIONS COMPILE	
GFILE ADA A	Command-line options plus defer generic instantiations

### Getting Information about an Input List Compilation

The compiler creates a file and places success or failure information about the compilation into it. Each line in the input list also appears in this file with a line that shows its compilation status. Source files or compilation units that compile successfully show a return code of zero for each compilation unit. Those that do not compile show the return code of the error that caused the failure. There are also descriptive messages where return codes do not provide enough information.

The return codes that can appear are:

Code	Explanation
0	Execution complete. No errors occurred.
4	Execution complete. Warnings were issued, but no errors occurred.
8	Source code errors, such as syntactic or semantic errors, were detected. Look for specific errors in the console listing.

The following is a brief example of an input list and the OUTPUT file that might result. This example uses VM/CMS file naming conventions; under MVS, the output follows MVS conventions.

### Sample Input List MYLIST INPUT A

```
&OPTIONS ASM
FILEONE ADA A
&OPTIONS DEF
FILETWO ADA A
&OPTIONS BLTZ
```



### Sample OUTPUT File

```
INPUT LIST processing MYLIST INPUT A - yyyy-mm-dd hh:mm:ss - options (options
&OPTIONS ASM
FILEONE ADA A
RC= 0 FILEONE ADA A1
&OPTIONS DEF
FILETWO ADA A
RC= 0 FILETWO ADA A1
&OPTIONS BLTZ
EVGDRV3561E ERROR IN INPUT LIST COMMAND SYNTAX
```

### VM/CMS File Name:

The name of the OUTPUT file takes the form *input\_list\_name* OUTPUT A, where *input\_list\_name* is the file name of the input list.

### MVS Data Set Name:

The name of the OUTPUT data set takes the form *qualifier*.OUTPUT(*input\_list\_name*), where *qualifier* is the current TSO profile prefix and *input\_list\_name* comes from the name of the input list. When the input list is a sequential data set, the second qualifier is used as the *input\_list\_name*. When the input list is a partitioned data set, the member name is used as the *input\_list\_name*.

## EVGADAI Cataloged Procedure

The EVGADAI cataloged procedure is used under MVS to batch compile multiple source files and/or compilation units with a single invocation of the compiler. It is the JCL equivalent of the ADA command with the INLIST option invoked from TSO. The EVGADAI cataloged procedure invokes the compiler with an input list file. Input list files are described in "Creating the Input List" on page 2-23. EVGADAI is used the same way as the EVGADAC cataloged procedure except for the following differences:

- The ADASRC symbolic variable specifies an input list data set instead of an Ada source data set.
- Information about each of the compilations is written to a SYSOUT data set whose class is specified by the SYSOUT symbolic variable.

A listing of the default EVGADAI cataloged procedure follows. It may have been changed by your system administrator when Ada/370 was installed on your system. The default EVGADAI procedure is supplied with the compiler in the SEVGPRO1 target library.

```
//EVGADAI      PROC CMPPRM=' ',MEMSIZE=8196K,
//              STPLIB='ADA.V1R4M0.SEVGMOD1',MAXTIME=60,
//              VIO=VIO,SYSDA=SYSALLDA,SYROUT='*'
//*
//*  ERASE ADA.INFO
//*
//              EXEC PGM=IEFBRI4
//ADAINFO      DD DSN=&USER..ADA.INFO,DISP=(MOD,DELETE),
//              SPACE=(1,1),UNIT=&SYSDA
//*****
//*                               INVOKE THE COMPILER
//*****
//STEP1        EXEC PGM=EVGINM,PARM='&CMPPRM',REGION=&MEMSIZE,
```



## ISPF/PDF Panels (MVS Only)

```
//      TIME=&MAXTIME,DYNAMNBR=65
//STEPLIB DD DSN=&STPLIB,DISP=SHR
//CONOUT DD SYSOUT=&SYSOUT,DCB=(LRECL=120,BLKSIZE=120)
//ADAIN  DD DSN=&ADASRC,DISP=SHR,FREE=END,DCB=BUFNO=4
//ADALIB DD DSN=&USER..ADA.LIBRARY,DISP=SHR
//ADAOUT DD SYSOUT=&SYSOUT,
//      DCB=(RECFM=VB,LRECL=259,BLKSIZE=3120)
//ADAINFO DD DSN=&USER..ADA.INFO,DISP=(NEW,PASS,CATLG),
//      DCB=(RECFM=VB,LRECL=512,BLKSIZE=3120,DSORG=PS),
//      SPACE=(80,(10,50)),UNIT=&SYSDA
//ADALIST DD DSN=&USER..LISTING,DISP=(MOD,CATLG,CATLG),
//      DCB=(RECFM=VBA,LRECL=259,BLKSIZE=3120,DSORG=PO,BUFNO=2),
//      SPACE=(132,(500,2000,20)),UNIT=&SYSDA
//ADAUT1 DD SPACE=(22528,(4,10)),
//      DCB=(RECFM=FB,BLKSIZE=4096,DSORG=DA,BUFNO=3),
//      UNIT=&SYSDA
//ADAUT2 DD SPACE=(132,(500,2000)),
//      DCB=(RECFM=VB,LRECL=136,BLKSIZE=3120,DSORG=PS,BUFNO=3),
//      UNIT=&VIO
//ADAUT3 DD SPACE=(22528,(4,10)),
//      DCB=(RECFM=FB,BLKSIZE=4096,DSORG=DA,BUFNO=3),
//      UNIT=&SYSDA
//ADAUT4 DD SPACE=(22528,(4,10)),
//      DCB=(RECFM=FB,BLKSIZE=4096,DSORG=DA,BUFNO=3),
//      UNIT=&SYSDA
```

---

## ISPF/PDF Panels (MVS Only)

Before using the ISPF/PDF panels, install:

1. The ISPF/PDF product for your MVS environment
2. The ISPF/PDF Ada/370 panels according to the instructions in the program directory.

The Ada/370 ISPF/PDF panels access the Ada/370 compiler, its tools and features. You can use the panels to compile, bind/link, and execute your Ada program in either the foreground or the background. You can also use them to invoke Ada tools for foreground interactive work.

The ISPF/PDF panels supports all common ISPF features including panel management, standard PF key settings, and field save between sessions. You can learn how to use the panels by invoking the tutorial panels which have brief descriptions of all the compiler and tool options.

When you encounter an error, you will see a short message on the top right corner of the panel. You can then press PF1 to obtain a longer message to help you respond to the error condition.

Some sample panels are shown in this book to provide you with an overview of how the ISPF/PDF panels work. However, you should consult the ISPF/PDF and Ada/370 tutorial panels for detailed information on how to use it.



## ISPF/PDF Panels Example

This section shows the ISPF/PDF panels that are used to compile, bind, link and execute an Ada/370 program in the foreground. To display the FOREGROUND ADA/370 COMPILE/RUN panel:

1. Select Foreground option from the ISPF-PDF PRIMARY OPTION MENU to get the FOREGROUND SELECTION PANEL
2. Select Ada/370 from the FOREGROUND SELECTION PANEL to get the FOREGROUND ADA/370 panel
3. Select Compile/Run from the FOREGROUND ADA/370 panel.

Then the following is displayed:

```

----- FOREGROUND ADA/370 COMPILE/RUN -----
COMMAND ==>

ISPF LIBRARY:  1
PROJECT ==>
GROUP  ==>      ==>      ==>      ==>
TYPE   ==>
MEMBER ==>      (Blank for member selection list)

OTHER PARTITIONED OR SEQUENTIAL DATA SET:
DATA SET NAME ==> 2

ADA LIBRARY:
DATA SET NAME ==> 3
DISPLAY/UPDATE ==> 4 (Yes,No)
INITIALIZE WORKING SUBLIB ==> 5 (Yes,No or number of units)

BROWSE LISTING ==> 6 (Yes,No)

CHANGE OPTIONS ==> 7 (Yes,No)
CURRENT OPTIONS ==> 8
==>
==>

```

Figure 2-3. Ada/370 Foreground Compile/Run Panel

- 1** Enter input file name for compilation. It is either a member of an ISPF library, a member of a partitioned data set or a sequential data set. You can specify the data set name by filling in Project, Group, Type, and Member fields or you can enter the data set name in **2**.
- 2** Enter the data set name when your input source is a member of a partitioned data set or a sequential data set. If you specify data sets in both **1** and **2**, the data set specified in this field will be used.
- 3** Enter the name of the Ada library file. The initial default data set name is *qualifier.ADA.LIBRARY*. If you change the name of library file for this entry then it becomes the default for this panel until you change it again.
- 4** Enter Yes when you want to view or modify the sublibraries listed in the Ada library file and it will display another panel to allow you to do this. The initial default is No. If you change this entry then it becomes the default for this panel until you change it again.



- 5** Enter Yes or a number to initialize your working sublibrary before compilation starts. The default is No.
- 6** Enter Yes to view all your compilation output listings after compilation or execution of the program. The initial default is No. If you change this entry then it becomes the default for this panel until you change it again.
- 7** Enter Yes when you want to use compiler options that are different from the ones that are shown in **8**. It will display the FOREGROUND ADA/370 COMPILE/RUN OPTIONS panel for you to modify the current compiler options. The initial default is No. If you change this entry then it becomes the default for this panel until you change it again.
- 8** Look at the current compiler options to decide whether you want to modify them by entering Yes in **7**. This is an output field, you can not enter any data here.

If you enter Yes for **7** in the previous panel, the following is displayed:

```

----- FOREGROUND ADA/370 COMPILE/RUN OPTIONS -----
COMMAND ==>

BIND ==>      (Yes,No)  LINK ==>      1 (Yes,No)  2 RUN ==>      (Yes,No)
CLEAR          ==>          (Yes,No)
EXPORT          ==>          (Yes,No)
INPUT LIST      ==>          (Yes,Max_number,No)
OPTIMIZE        ==>          (Yes,Autoinline,No)
DEBUGGER INFORMATION ==>      (Yes,No)
MAXIMUM ERRORS  ==>          (1 to 32767)
SOURCE LISTING  ==>          (Yes,Conditional,No)
ASSEMBLY LISTING ==>          (Yes,NOSys,NOGen,NONE)
CROSS REFERENCE LISTING ==>    (Yes,Full,No)
  SORT BY UNIT  ==>          (Yes,No)
CALL GRAPH LISTING ==>        (Yes,No)
LINKAGE MAP     ==>          (Yes,No)
COMPILER TRACE  ==>          (Yes,No)
SUPPRESS RUN TIME CHECKS ==>    (Yes,Elaboration,No)
SUPPRESS LINE INFORMATION ==>  (Yes,No)
GENERIC INSTANTIATION MODE ==>  (IMmediate,Deferred,INline)
COMPILE GENERIC INSTANCES ==>  (Yes,No)
SHARED GENERICS REPORT ==>      (Yes,No)
PASSIVE TASKING ==>          (Yes,No)
USED: ADAIN ==>      (Yes,No)  USED: ADALIB ==>      (Yes,No)

```

Figure 2-4. Ada/370 Foreground Compile/Run Options Panel

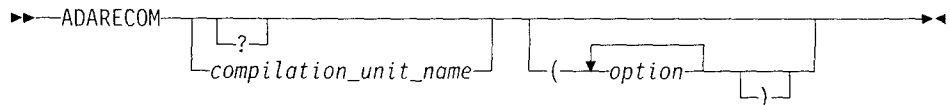
- 1** Look at the default values for the compiler options that were set from your last compilation using the ISPF/PDF panels. Overtyping any of these options for this compilation and they will also become your default compilation options for subsequent ISPF foreground compilations.
- 2** Look at the possible values that you can type in **1**. Use the Ada/370 tutorial panels to obtain explanations on these compiler options.

Press the enter key and the compilation and/or execution of the Ada/370 program will begin.



## Generating Recompilation Lists

When a library consists of many compilation units that depend on (possibly multiple levels of) **with** statements, it can be difficult to determine the proper compilation order when the specification of a unit needs to be recompiled. You can use the ADARECOM command to help you identify those source files that require recompilation. The compilation units must have already been compiled into a library. The ADARECOM command reads this library to create a recompilation list of the units that depend upon the specified unit. This recompilation list is stored in a file called *comp\_unit* INLIST A under VM/CMS or *qualifier*.INLIST(*comp\_unit*) under MVS. *comp\_unit* is the name of the compilation unit with underscores removed and truncated to eight characters. The same file can also be used as input to the INLIST option to the ADA command. For more on this subject, see "Compiling Multiple Source Files" on page 2-22.



The ADARECOM command generates a recompilation order list of source files containing the compilation units that depend on the specified *compilation\_unit\_name*. To use ADARECOM, you must have compiled the compilation units into a specified library at least once. For any compilation unit that contains instance subunits that have not been compiled or are out-of-date, that compilation unit name will also appear in the recompilation order list with a preceding &OPTIONS DICOPILE line and followed by an &OPTIONS COMPILE line. See "Creating the Input List" on page 2-23 for information on the format of the input list.

For example, if instance subunits are reported as missing or out-of-date at bind time, you can also use the ADARECOM command to create a list of compilation units that need to be compiled with the DICOPILE option.

The ? option provides syntax information on the ADARECOM command.

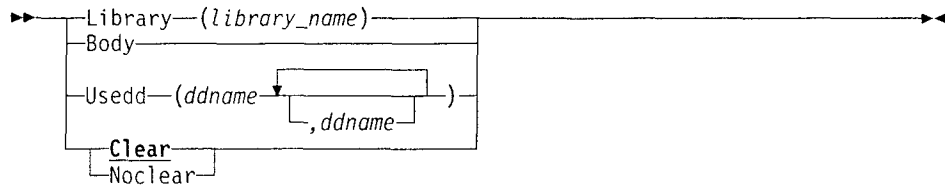
### Notes:

1. If you move an Ada compilation unit to a new source file, you must modify the recompilation list.
2. ADARECOM produces a correct source file list only if each file contains a single compilation unit.
3. For ADARECOM to correctly generate the recompilation list, the association between file names and compilation units cannot change between the time they are compiled and the ADARECOM invocation.

Between the time you make the recompilation list and the time you perform the recompilation, do not change the names of the source files, separate the specification and body into different files, or perform any other change that alters the relationship between that file and the compilation unit or units it contains. You can edit the recompilation list after running ADARECOM to make changes to the recompilation order list.



The *option* modifier in the preceding diagram expands to this:



Precede the list of options by a blank space and a left parenthesis, and separate them from each other by blank spaces. A closing parenthesis is optional.

### Library

Specifies the name of the Ada library file that ADARECOM is to read.

#### Under VM/CMS

You can provide the *library\_name* variable in either of two formats. The preferred format is *file\_name file\_type file\_mode*. The other format is *file\_mode:file\_name.file\_type*. In both formats, if you specify only the file name, *file\_type* defaults to LIBRARY and *file\_mode* defaults to "\*". If you do not specify a library file, ADA searches for ADA LIBRARY \*. The recompilation list goes into a file called *comp\_unit* INLIST A, where *comp\_unit* is the name of the compilation unit with underscores removed and truncated to eight characters.

The command

```
ADARECOM MY_PROG (L(MYLIB LIBRARY))
```

generates a recompilation list that includes all units within MYLIB LIBRARY that must be recompiled if you recompile MY\_PROG, also found in that library.

#### Under MVS

If you do not specify *lib\_name*, the default is *qualifier.ADA.LIBRARY*.

The recompilation list goes into a file called *qualifier*.INLIST(*comp\_unit*), where *qualifier* is your TSO profile prefix and *comp\_unit* is the name of the compilation unit with underscores removed and truncated to eight characters.

The command

```
ADARECOM MY_PROG (L(MYLIB))
```

generates a recompilation list that includes all units within *qualifier*.MYLIB that must be recompiled if you recompile MY\_PROG, also found in that library.

**Body** If you specify the Body option, ADARECOM assumes that *comp\_unit\_name* refers to the body of a compilation unit. By default, *comp\_unit\_name* refers to the specification of a compilation unit.

### Clear | Noclear

Enables automatic clearing of the terminal screen before processing begins. The Noclear option suppresses automatic clearing of the terminal screen. Clear is the default.

### Usedd (MVS Only)

Specifies the DD name of the data set that you have preallocated to the standard DD name for the ADARECOM command. The data set is normally allocated by the ADARECOM command.



The ADALIB option modifier is the standard ADARECOM command DD name of the data set that you can preallocate. ADALIB is the Ada library data set. Specifying the `LIBRARY(library_name)` option and `USED(ADALIB)` causes the `Used` option to take precedence.

See the "Using Ada/370 Compiler and Tools" on page 1-4 for additional guidelines on using the Ada/370 tools.

## Analyzing Source Dependencies: the ADADEP Command

This command determines, from a given list of Ada source files, a compilation order for those files. The source code is analyzed to determine the dependencies created by context clauses. The output file that this command generates can be used as a compiler input list to compile the files in the correct order.

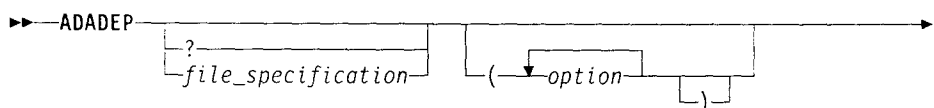
### Notes:

1. If you do not defer generic instantiations, the generic bodies must be compiled before the associated generic is instantiated.
2. Some kinds of dependencies are difficult to determine through source analysis alone. For example, if a package depends on another package containing generic units, generic specifications, or subprograms that have pragma `INLINE` applied to them, the dependencies are different if the first package instantiates the generics or calls any subprograms that are expanded inline.

The ADADEP command assumes that all possible dependencies introduced by generic units really occur. If this assumption is incorrect, ADADEP may not find a correct compilation order even when one exists. If you know that a correct compilation order exists, use the `GENERIC` option to override the assumptions ADADEP makes.

3. No dependencies are generated for units defined by the language standard (for example `TEXT_IO` or `CALENDAR`) or supplied by Ada/370 (for example `IMS` or `COMMAND_LINE`). The ADADEP command does not report an error when the source for these units is not present in the specified source files.

## Syntax



? displays a help message for ADADEP. It has the same function as the `HELp` option. If specified, it must be the first word after the command name, and the ADADEP command ignores the remainder of the command line.

The *file\_specification* is the name of an input list. It must contain at least one Ada source file name. It can also contain Ada comments: ADADEP ignores everything after `--` on a line. The ADADEP defaults for *file\_specification* are shown in Table 2-3 on page 2-34.

Also see the "Using Ada/370 Compiler and Tools" on page 1-4 for additional guidelines on using Ada/370 tools.



**Under VM/CMS:**

Specify the *file\_specification* in the standard format of *file\_name file\_type file\_mode*. In file specifications within the input list, *file\_type* and *file\_mode* default to ADA and \* respectively.

If *file\_specification* is not specified, the default ADADEP INLIST \* is used.

**Under MVS:**

Specify the data set name partially qualified (the default) or fully qualified with single quotation marks.

You can omit some or all of the *file\_specification*. The ADADEP command supplies its own default, which is *qualifier.ADADEP.INLIST*.

Table 2-3. ADADEP Defaults for File Specifications

Purpose	VM/CMS			MVS (TSO)
	File Name	File Type	File Mode	Data-Set Name
Input list file	ADADEP	INLIST	*	<i>qualifier.ADADEP.INLIST</i>
Output file	ADADEP	OUTPUT	A	<i>qualifier.ADADEP.OUTPUT</i>

**ADADEP Options**

Specify options in a list separated by blanks, and precede the beginning of the option list with a blank space and a left parentheses. Some option names are followed by one or more optional modifiers, enclosed in parentheses. Options with more than one modifier are separated by a comma. None of the options are in effect unless they are specified. By default, the output will be displayed on the terminal.

All options can be specified in any unique minimum abbreviation form. For example, Continue and CONTinue are equivalent. The following table lists the options for the ADADEP command:

Table 2-4 (Page 1 of 3). Options for the ADADEP Command

Name	Description	VM	MVS
Continue	Allows dependency analysis to continue even if the source files in the input list contain syntax or other errors. An incorrect compilation list may result because of missing or incorrect information within the Ada source files. This option implicitly puts the Missing option into effect, so the Missing option cannot be specified with the Continue option.	√	√
Exec	Indicates that the input list is in the CMS EXEC format created by the LISTFILE command with the EXEC option. This type of input list must not contain Ada-style comments.	√	



Table 2-4 (Page 2 of 3). Options for the ADADEP Command

Name	Description	VM	MVS
Fast	<p>Does a fast analysis of the source files using the following rules:</p> <ul style="list-style-type: none"> <li>• Only the first compilation unit in any file will be processed. If a file contains more than one compilation unit, ADADEP behaves as if the other units do not exist, and they may be reported as missing.</li> <li>• Generic units inside nongeneric units will not be detected because Fast implicitly puts the Generic option into effect; therefore, you cannot specify the Generic option with the Fast option.</li> <li>• Some syntax errors may not be detected because only a part of each file is examined.</li> </ul> <p>This option is useful to analyze a set of files where each file contains only one compilation unit. In this case, ADADEP only looks at the first part of each file to determine dependencies and can produce a faster analysis. Fast cannot be used with Inline.</p>	✓	✓
Generic	Causes ADADEP to ignore dependencies created by nongeneric units that contain generic units. It does not affect dependencies created by generic units.	✓	✓
Help	Displays help text. Equivalent to specifying ? instead of a file specification.	✓	✓
Inline	Causes ADADEP to ignore dependencies caused by pragma INLINE. Inline cannot be used with Fast.	✓	✓
Missing	Indicates that some units may be missing and that the ADADEP command is to assume that missing units do not introduce any dependencies of interest.	✓	✓
Output [file_specification]	<p>Designates the file or data set to contain the output. If you specify an MVS partitioned data set, it must already exist and must have the following characteristics:</p> <p>Record Format: VB Record Length: 259 Block Size : 13030</p> <p>The output is in the compiler's input-list format, enabling you to quickly compile the files in the order that ADADEP determines. If this option is not specified, the output is displayed on the display screen.</p>	✓	✓



Table 2-4 (Page 3 of 3). Options for the ADADEP Command

Name	Description	VM	MVS
Prompt	Prompts for confirmation to overwrite the output file if that file already exists. This option is not allowed when ADADEP is run in batch mode or when the Replace option is also specified. If Prompt is specified when ADADEP is running in TSO emulation batch, the Prompt option is ignored. You must also specify the Output option with this option.	✓	✓
Replace	Replaces the output file if it already exists. This option requires Output to be also specified; it is not allowed in combination with the Prompt option.	✓	✓
UNit <i>unit_name</i>	Indicates that dependency information need only be generated starting from the given unit. The unit name may start with <b>lib/</b> or <b>sec/</b> . If it does not, dependency information is generated for both the specification and body of the given unit if both are present in the source files.	✓	✓
USedd(ADAIN)	Specifies the DD name of the data set that has already been preallocated. If an input list file and USedd are both specified, the USedd option takes precedence.		✓
Verbose	Causes ADADEP to display progress messages as it analyzes dependencies. You may want to do this analysis if you are analyzing a large number of files and want to see where ADADEP is in its analysis.	✓	✓

## Sample Invocation Commands

### VM/CMS

1. ADADEP program (Generic Missing)  
Analyzes PROGRAM INLIST \*, ignoring dependencies created by missing units and nongeneric units that contain generic units. The output is displayed on the display screen.
2. ADADEP mylist list2 k (Unit(main) Exec Prompt Output(myout))  
Analyzes files named in MYLIST LIST2 K (which is in CMS EXEC format). The output starts with the unit MAIN, and is written into MYOUT OUTPUT A, with a request for confirmation to delete the output file if it already exists.
3. ADADEP  
Analyzes the files named in ADADEP INLIST \* and displays the results on the display screen.



**MVS (TSO)**

1. ADADEP program (Generic Missing)  
Analyzes '*qualifier*.PROGRAM', ignoring dependencies created by missing units and nongeneric units that contain generic units. The output is displayed on the display screen.
2. ADADEP 'user1.mylist.list2' (Unit(main) Replace Output(myout))  
Analyzes files named in 'USER1.MYLIST.LIST2'. The output starts with the unit MAIN, and is written into '*qualifier*.MYOUT', overwriting it if it already exists.
3. ADADEP  
Analyzes the data sets named in '*qualifier*.ADADEP.INLIST' and displays the results on the display screen.

**Temporary Files**

Under VM/CMS, the ADADEP command creates temporary files with the name A\$D\$E\$P\$ and different file types. Do not use this name for your own files.

Under MVS, the ADADEP command creates temporary files with the name *user\_id*.A\$D\$E\$P\$.*name*, using different names for the last component. Do not use names of this form for your own data sets.

---

**Compilation of Generic Units**

IBM Ada/370 supports separately compiled generic units. You can compile a generic specification in one file and its generic body in a separate file.

When you are not inlining generic instantiations, Ada/370 compiles a generic instantiation similar to an Ada body stub with the generic instance treated as an Ada subunit, referred to as the instance subunit. See "Brief Unit Report List Format" on page 5-15 for a description of the name of the instance subunit.

If you compile a generic body, you can recompile the non-inlined instances of the generic unit by using the DIcompile option. The compilation must be applied to each compilation unit that instantiates the generic unit, but only the out-of-date instantiations will be recompiled. The rest of the compilation unit is unaffected. For more information on the compiling and instantiation of generic units, see Chapter 12 of the LRM. You can use the ADARECOM command to generate an input list of compilation units that have out-of-date or missing instance subunits.

If you plan to instantiate and compile any generic unit with your compilation unit, compile the generic body before you attempt to instantiate the generic unit. The generic body must be compiled and visible in the library before the instantiation can occur. Otherwise, the compiler issues a warning and defers the instantiation.

**Note:** You can separately compile subunits of a generic unit.

**Deferring Instantiation of Generic Units**

You can defer the instantiation of generic units. For example, to compile a generic instantiation before compiling the body of the generic template, do the following:

1. Compile the specification of the generic.
2. Compile the compilation unit that has the instance of the generic unit, with the INSTantiation (Deferred) option.



3. When you have compiled the body of the generic unit, you can compile the instance subunit by using the DIcompile option on the compilation unit in step 2.

In the last step, the compilation is only performed on those deferred instances or out-of-date instance subunits. The compilation unit itself will not be compiled or updated in the library.

### Compilation Examples

Eight compilation examples are shown with the following Ada source:

#### *Ada Source Programs*

GENSPEC ADA

```
generic
  type Item is private;
  procedure Gen (X,Y : in out Item);
```

GENBODY ADA

```
procedure Gen (X,Y : in out Item) is
  Tmp : Item := X;
begin
  X := Y;
  Y := Tmp;
end Gen;
```

GEN2 ADA

```
generic
  procedure Gen2;
  pragma inline_generic (Gen2);

  procedure Gen2 is
  begin
    null;
  end Gen2;
```

TEST ADA

```
with Gen;
procedure Test_Compilation_Unit is
  A,B : integer := 0;
  procedure Swap is new Gen (integer);
begin
  Swap (A,B);
end Test_Compilation_Unit;
```

TEST2 ADA

```
with Gen; with Gen2;
procedure Test2_Compilation_Unit is
  A,B : integer := 0;
  procedure Swap is new Gen (integer);
  pragma inline_generic (Swap);
```



```

    procedure Do_Nothing is new Gen2;
begin
    Swap (A,B);
    Do_Nothing;
end Test2_Compilation_Unit;

```

### Compilation Sequences

1.

```

ada GENSPEC  (create
ada GENBODY
ada TEST     (main instantiation(inline)

```

The generic instance is expanded inline. The bind is successful without any additional compilations.

2.

```

ada GENSPEC  (create
ada TEST     (instantiation(immediate)
ada GENBODY
ada test_compilation_unit  (dicompile
ada test_compilation_unit  (bind

```

Because the generic body has not been compiled and TEST has used immediate, a warning is generated and the instantiation is deferred. The instance subunit is then compiled (by using the dicompile) before binding.

3.

```

ada GENSPEC  (create
ada GENBODY
ada TEST     (main instantiation(immediate)

```

Because the generic body is available (compiled) and TEST is compiled with immediate, no extra compilations are required for a successful bind.

4.

```

ada GENSPEC  (create
ada GENBODY
ada TEST     (instantiation(deferred)
ada test_compilation_unit  (dicompile
ada test_compilation_unit  (bind

```

Because TEST is compiled with deferred, the instance subunit is expected to be compiled by a separate invocation of the compiler.



5.

```
ada GENSPEC  (create
ada TEST    (instantiation(deferred)
ada GENBODY
ada test_compilation_unit (bind
```

The bind fails since the instance subunit `test_compilation_unit.swap__1` has not been compiled. `dicompile` on `test_compilation_unit` should have been done prior to bind.

**Note:** The number 1 in `test_compilation_unit.swap__1` is generated by the compiler.

6.

```
ada GENSPEC  (create
ada GENBODY
ada TEST    (instantiation(immediate)
ada GENBODY
ada test_compilation_unit  (bind
ada test_compilation_unit  (dicompile
ada test_compilation_unit  (bind
```

The first compilation of `TEST` causes the instance subunit `test_compilation_unit.swap__1` to be compiled at the same time as `TEST`. `GENBODY` is then recompiled. The first attempt to bind fails because `test_compilation_unit.swap__1` is now out-of-date. After `test_compilation_unit.swap__1` is recompiled, the bind is successful.

7.

```
ada GENSPEC  (create
ada GENBODY
ada GEN2
ada TEST2    (instantiation(deferred)
```

Because of the pragmas in `GEN2` and `TEST2`, the instantiations for `Swap` and `Do_Nothing` are inlined by the instantiation method even though deferred is used. If either the generic template body, `Gen` or `Gen2`, changes, `TEST2` must be recompiled.

---

## Compilation of Passive Tasks

The Ada/370 compiler recognizes tasks that can be transformed into passive tasks without using special pragmas in the source program. The Ada/370 compiler transforms these tasks into passive tasks only when you specify the `Passive` option when compiling the specification and body of a task. You can use the `Optimize` option with the `Passive` option for a compilation unit. See the *Ada/370 Programmer's Guide* for more information on passive tasking.



## LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.



---

## Chapter 3. Binding and Linking an Ada Program

This chapter describes how you can invoke the IBM Ada/370 binder or the linkage editor, under the VM/CMS, and the MVS TSO, or batch environments. A description of binding and linking a collection of Ada exported routines is also provided.

Before reading this chapter, you should be familiar with the use of the ADA command and its options.

---

### Overview of Binding and Linking

A special Ada linker called the IBM Ada/370 binder partially links object modules produced by the IBM Ada/370 compiler and outputs them as a standard IBM object module. This partially linked object module is further processed by the system linker or loader to produce an executable load module. The IBM Ada/370 binder provides full support of Ada requirements for symbol naming. It also drastically reduces the number of external definitions and references that must be processed by the system linker or loader.

An Ada program can use pragma INTERFACE to call subprograms written in a programming language other than Ada. The system linker puts the standard-format object modules produced for these subprograms into the executable load module it creates for an Ada program. For information on using the transporter to import/export non-Ada object code into the Ada library, see "The ADATRANS Command" on page 5-29.

The IBM Ada/370 binder also includes run time environment routines as part of its output.

---

### Binding IBM Ada/370 Programs

This section describes the different ways you can invoke the binding step. Most often you will use the ADA command to invoke the Ada/370 binder. If you run MVS, you can also use JCL or the EVGADAB procedure to bind programs in the batch environment.

**Note:** If ISPF/PDF is installed under MVS, you can invoke the binder by using the ISPF/PDF panels described in "ISPF/PDF Panels (MVS Only)" on page 2-28.

### Using the ADA Command to Bind

To invoke the IBM Ada/370 binder, compile an Ada main program using the MAIn option of the ADA command. The binder produces a link map describing the contents of the partially linked object module it generates. The link map provides you with detailed information about the run time memory locations of the various pieces of code that make up your program.

Another option of the ADA command, Bind, causes IBM Ada/370 to bypass the compilation step. You can take a compilation unit that you have already compiled as a library unit and bind it as a main program.



## Using JCL to Bind a Main Program (MVS only)

The GGenerate option of the ADA command takes binder output and uses system utilities to generate a load module.

For more information on the MAIn, Bind, and GGenerate options of the ADA command, see "The Compiler Options" on page 2-3.

You must rebind your main program when you recompile any Ada compilation units used in the program. You do not have to rebind the program if you recompile non-Ada routines that your Ada program calls, but you still have to link the program again with the linkage editor or loader.

## Using Job Control Language to Bind a Main Program (MVS only)

This section describes how to invoke the binder as a MVS batch job by using Job Control Language (JCL). For information on how to invoke the compiler using JCL, see "Compiling a Program with Job Control Language (JCL)" on page 2-18.

The EVGADAB cataloged procedure invokes the IBM Ada/370 binder to bind an Ada main program that has been compiled using the IBM Ada/370 compiler. The output of the binder is a System/370 relocatable object data set. You can submit this data set to the linkage editor to generate an executable load module.

```
//MYPROG JOB ,',MSGCLASS=D,MSGLEVEL=(1,1),NOTIFY=USER1',  
// CLASS=A  
//*  
//* PURPOSE: TO RUN THE ADA BINDER  
//*  
//BIND EXEC PROC=EVGADAB,UNIT=HELLO,  
// USER=USER1
```

Figure 3-1. Using the EVGADAB Cataloged Procedure to Invoke the Binder

The preceding example shows a job, called MYPROG, which binds the Ada main compilation unit HELLO. The user identifier USER1 is specified with the USER variable. This variable is used as a high-level qualifier to construct data set names for the compiler, such as USER1.ADA.LIBRARY. This library is the default Ada library. Your job card will probably be different, because it depends on your site's conventions. This example assumes that the EVGADAB cataloged procedure is visible to the jobstep.

As this job is executed, the compiler creates relocatable object code in USER1.OBJ(HELLO). This object code was generated for the Ada main compilation unit called HELLO.



A sample of the EVGADAB cataloged procedure appears in Figure 3-2. The exact location of EVGADAB may depend on your site's conventions, but is installed in SEVGPRO1 by default.

```
//EVGADAB      PROC BNDPRM=' ',MEMSIZE=2048K,
//            STPLIB='ADA.V1R4M0.SEVGMOD1',MAXTIME=60,
//            VIO=VIO,SYSDA=SYSALLDA,SYSOUT='*',UNIT=' '
//*****
//*                               ERASE ADA.INFO DATASET
//*****
//            EXEC PGM=IEFBR14
//ADAINFO DD   DSN=&USER..ADA.INFO,DISP=(MOD,DELETE),
//            SPACE=(1,1),UNIT=&SYSDA
//*****
//*                               INVOKE THE BINDER
//*****
//STEP1      EXEC PGM=EVGBIND,PARM='&UNIT. ( &BNDPRM',REGION=&MEMSIZE,
//            TIME=&MAXTIME,DYNAMNBR=65,COND=(4,LT)
//STEPLIB DD   DSN=&STPLIB,DISP=SHR
//CONOUT DD   SYSOUT=&SYSOUT,DCB=(LRECL=120,BLKSIZE=120)
//ADALIB DD   DSN=&USER..ADA.LIBRARY,DISP=SHR
//ADAINFO DD   DSN=&USER..ADA.INFO,DISP=(NEW,PASS,CATLG),
//            DCB=(RECFM=VB,LRECL=512,BLKSIZE=3120,DSORG=PS),
//            SPACE=(80,(10,50)),UNIT=&SYSDA
//ADAOBJ DD   DSN=&USER..OBJ,DISP=(MOD,CATLG,CATLG),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120,DSORG=PO,BUFNO=4),
//            SPACE=(80,(16000,16000,20)),UNIT=&SYSDA
//ADAMAP DD   DSN=&USER..ADAMAP,DISP=(MOD,CATLG,CATLG),
//            DCB=(RECFM=VB,LRECL=1023,BLKSIZE=3120,DSORG=PO,BUFNO=2),
//            SPACE=(132,(1000,2000,20)),UNIT=&SYSDA
//ADADMAP DD   DSN=&USER..DEBUGMAP,DISP=(MOD,CATLG,CATLG),
//            DCB=(RECFM=VB,LRECL=1023,BLKSIZE=3120,DSORG=PO,BUFNO=2),
//            SPACE=(132,(1000,2000,20)),UNIT=&SYSDA
//ADALIST DD   DSN=&USER..LISTING,DISP=(MOD,CATLG,CATLG),
//            DCB=(RECFM=VBA,LRECL=259,BLKSIZE=3120,DSORG=PO,BUFNO=2),
//            SPACE=(132,(500,2000,20)),UNIT=&SYSDA
//ADAUT1 DD   SPACE=(4096,(50,100)),
//            DCB=(RECFM=FB,BLKSIZE=4096,DSORG=DA,BUFNO=3),
//            UNIT=&SYSDA
//ADAUT2 DD   SPACE=(3120,(30,90)),
//            DCB=(RECFM=VB,LRECL=258,BLKSIZE=3120,DSORG=PS,BUFNO=3),
//            UNIT=&VIO
//ADAUT3 DD   SPACE=(4096,(30,60)),
//            DCB=(RECFM=FB,BLKSIZE=4096,DSORG=DA,BUFNO=3),
//            UNIT=&SYSDA
//ADAUT4 DD   SPACE=(4096,(30,60)),
//            DCB=(RECFM=FB,BLKSIZE=4096,DSORG=DA,BUFNO=3),
//            UNIT=&SYSDA
//ADAUT5 DD   SPACE=(3120,(30,60)),
//            DCB=(RECFM=VB,LRECL=258,BLKSIZE=3120,DSORG=PS),
//            UNIT=&SYSDA
```

Figure 3-2. EVGADAB Cataloged Procedure



### Symbolic Variables for EVGADAB Cataloged Procedure

The EVGADAB cataloged procedure includes several symbolic JCL substitution variables you can modify to specify the various options available.

Symbolic Variable	Description
----------------------	-------------

BNDPRM	Specifies options to the binder in the PARM field. A list of options you can specify in BNDPRM, along with a syntax diagram, appears in "Binder Options for Use with EVGADAB Cataloged Procedure" on page 3-5.
MAXTIME	Sets a maximum amount of time for the binder job step to run (using the TIME parameter on a JCL EXEC card). The default value is 60 minutes, but it can be changed when you install the cataloged procedure.
MEMSIZE	Specifies the amount of memory in which to run the binder job step (using the REGION option on a JCL EXEC card). The greater the memory, the better the binder's performance. The default value is 8196 KB, but this value may be changed when you install the cataloged procedure.
STPLIB	Indicates the data set name of the partitioned data set that contains the binder module, EVGBIND. The default is the load library ADA.V1R4M0.SEVGMOD1, but it may have been changed when the cataloged procedure was installed.
SYSDA	Specifies the UNIT for permanent data set allocations. The default is SYSALLDA, but you can change this default depending on your site's conventions.
SYSOUT	Identifies the output class for the binder output. The default is *, but you can change this default depending on your site's conventions.
UNIT	Indicates the compilation unit to be bound. You must specify this variable.
USER	Indicates the high-level qualifier the binder requires to build data set names. You must specify this variable. It is common to set it to your TSO profile prefix.
VIO	Specifies the UNIT for temporary data set allocations. The default is VIO, but you can change this default depending on your site's conventions.



## Binder Options for Use with EVGADAB Cataloged Procedure

This section defines the standard options for the BNDRPM symbolic substitution variable. This variable, found in STEP1 of the EVGADAB cataloged procedure, specifies options to the binder in the PARM field.

The binder options have the following syntax:

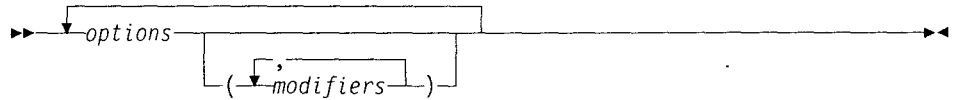


Table 3-1 lists the valid options, their modifiers, and defaults. Descriptions of the options appear on the pages shown.

Table 3-1. Binder Options			
Option	Default	Function	Page
Asm	NOAsm	Assembler listing.	2-5
Clear	Clear	Enable automatic clearing of the terminal screen by the binder.	2-7
NOClear		Disable automatic clearing of the terminal screen by the binder.	
DDnames old_name=new_name		Specify the Data Description (DD) names that identify the data sets used by the compiler and binder. It must include at least one modifier.	2-7
DEbug	NODEbug	Output information for debugging.	2-8
Export	NOExport	Specifies that a non-Ada main program is to be linked with an output object file.	2-10
Map	NOMap	Produce linkage map during binding.	2-14
Shared	NOShared	Used with Bind option, cause a report that describes the sharing of generics to be generated in the binder listing file.	2-15
Trace	NOTrace	Display diagnostic messages from the compiler. Only for use in submitting problems to IBM.	2-16

## Binding Multiple Main Programs

To bind multiple main programs in one invocation of the Ada/370 binder, you must first create an input list. An input list is a file containing a list of the Ada unit names of program units to be bound. See "Creating the Input List" on page 3-6 for the syntax of each line of the input list. You can then bind the main program units whose names are in this input list by using the ADA command with the INList and Bind options on either VM/CMS or MVS, or use the EVGADABI cataloged procedure in an MVS JCL batch job. See "EVGADABI Cataloged Procedure" on page 3-9.

The names of main program units appear in the input list along with other information that controls the binding process. Along with producing bound object code for each main unit in the list, the Ada/370 binder produces a file that contains information on each successful or failed binding of a main program. If you use a



binder option that produces a listing (Asm or Shared), a linkage map (Map), or a debug map (DEbug), the binder produces separate files for each main program.

If the binder detects errors during the processing of any main program in the list, it goes on to process the next main program. There may be cases, especially with a large input list, where it is not advisable to continue through the entire input list when multiple main programs fail to bind. The INList option has a modifier that allows you to specify the maximum number of main program bind failures before the binding process stops.

#### **Under VM/CMS:**

For example, the command string

```
ADA MYLIST (B INL(6)
```

binds the main programs in the input list MYLIST INLIST \*, setting the failure limit at six.

#### **Under MVS:**

For example, the command string

```
ADA MYLIST.INLIST (B INL(6)
```

binds the main programs in the input list *qualifier*.MYLIST.INLIST, setting the failure limit at six. *qualifier* is the current TSO profile prefix.

When the ADA command is used on MVS:

- With the Bind and INList options, each successfully bound main program is processed by the linkage editor (unless the NOGenerate option is specified) after the termination of the Ada Binder input list processing.
- With the Bind, INList, and Run options, each successfully bound main program is first processed by the linkage editor and then executed after the termination of the Ada Binder.

When the ADA command is used on VM/CMS:

- With the Bind, INList, and GGenerate options, each successfully bound main program is processed by the VM/CMS LOAD and GENMOD commands after the termination of the Ada binder.
- With the Bind, INList, and Run options, each successfully bound main program is first processed by the VM/CMS LOAD command and then executed after the termination of the Ada binder.
- With the Bind, INList, GGenerate, and Run options, each successfully bound main program is first processed by the VM/CMS LOAD and GENMOD commands and then executed after the termination of the Ada binder.

## **Creating the Input List**

An input list can contain four kinds of lines. The syntax for each line of the input list is:





Main-unit-name lines specify main programs to be bound. Bind-time option lines specify binder options to be used. Additionally, an input list may contain comment lines and blank lines which have no effect on the binding process.

The rules for creating the input list are:

- Place only one main unit name or one bind-time option on a line.
- Do not place comments on the same line with other information.
- The binder ignores blank lines.
- Main unit names do not have to start in the first column.

This VM/CMS input list follows the rules correctly:

```
MAIN_A
    MAIN_C
* THIS IS A COMMENT
MAIN_E

&OPTIONS MAP ASM
    MAIN_F
&OPTIONS NOMAP
    MAIN_G
```

### Embedding Bind-Time Options in the Input List

The &OPTIONS line is used to place bind-time options and input list options within the input list. It should precede the main program name or names to which it applies. The options that are set when you invoke the binder apply to each main program until overridden by an option embedded in the input list. Options that appear in the input list apply to all of the following main programs until overridden by a subsequent embedded option. A specific option can appear more than once in an input list. The options can have the same modifiers as when you specify them on the command line. The syntax for the &OPTIONS line is:

→&OPTIONS—binder\_option—input\_list\_option→

The *binder\_options* that you can embed in input lists are:

```
Asm
Bind
Clear
DEBug
Export
Inlist
Map
NOAsm
NOBind
NOClear
NODebug
NOExport
NOMap
NOShared
Shared.
```

**Note:** Inlist in an &OPTIONS line is only used to change the maximum number of failures allowed during input list processing.



There is only one valid *input\_list\_option*, DEFault. It causes the binder to reset all options to their states as set by the ADA command or the EVGADABl cataloged procedure invocation JCL statement.

The following example shows an input list with embedded options, along with descriptions of how the options change:

Input List	Options Used for Compilation
MAIN_A &OPTIONS MAP	Command-line options
MAIN_B &OPTIONS NOMAP	Command-line options and Map
MAIN_C &OPTIONS DEFAULT	Command-line options and NOMap
MAIN_D &OPTIONS ASM	Command-line options only
MAIN_D &OPTIONS NOBIND	Command-line options and Asm
MAIN_E  &OPTIONS BIND	Command-line options, Asm, and NOBind (main program Main_E is skipped)
MAIN_F	Command-line options and Asm

### Getting Information about an Input List Bind

The binder creates a file and places success or failure information about each bind into it. Each line in the input list also appears in this file with a line that shows the bind status. Main programs that bind successfully show a return code of zero. Those that do not bind successfully show a return code indicating the nature of the error. There are also descriptive messages when return codes do not provide enough information.

The return codes that can appear are:

Code	Explanation
0	Execution complete. No errors occurred.
4	Execution complete. Warnings were issued, but no errors occurred.
8	Source code errors, such as syntactic or semantic errors, were detected. Look for specific errors in the console listing.

The following is a brief example of an input list and the OUTPUT file that might result.

#### Sample Input List:

```
&OPTIONS ASM
MAIN_A
&OPTIONS MAP
MAIN_B
&OPTIONS DEFAULT
MAIN_C
&OPTIONS BUZZ
```



#### Sample OUTPUT File:

```
INPUT LIST processing SAMPLE  INLIST  A1 - yyyy-mm-dd hh:mm:ss - options
&OPTIONS ASM
MAIN_A
RC= 0 MAIN_A
&OPTIONS MAP
MAIN_B
RC= 0 MAIN_B
&OPTIONS DEFAULT
MAIN_C
RC= 0 MAIN_C
&OPTIONS BUZZ
EVGDRV3561E >>> Error in input list command syntax
```

**VM/CMS File Name:** The name of the OUTPUT file takes the form *input\_list\_name* OUTPUT A, where *input\_list\_name* is the file name of the input list.

**MVS Data Set Name:** The name of the OUTPUT file takes the form *qualifier.OUTPUT(input\_list\_name)*, where *qualifier* is the current TSO profile prefix and *input\_list\_name* comes from the name of the input list. When the input list is a sequential data set, the second qualifier is used as the *input\_list\_name*. When the input list is a partitioned data set, the member name is used as the *input\_list\_name*.

### EVGADABI Cataloged Procedure

The EVGADABI cataloged procedure is used under MVS to batch bind multiple main programs with a single invocation of the binder. It is the JCL equivalent of the ADA command with the INLIST and Bind options invoked from TSO. The EVGADABI cataloged procedure invokes the binder with an input list file. Input lists are described on page 3-6. EVGADABI is used the same way as the EVGADAB cataloged procedure except for the following differences:

- The UNIT symbolic variable is not used.
- The ADASRC symbolic variable specifies an input list data set.
- Information about each of the main program binds is written to a SYSOUT data set whose class is specified by the SYSOUT symbolic variable.

The binder options for EVGADAB, described on page 3-5, also apply to EVGADABI.

A listing of the default EVGADABI cataloged procedure follows. It may have been changed by your system administrator when Ada/370 was installed on your system. The default EVGADABI procedure is supplied in the SEVGPRO1 target library.



```

//EVGADABI PROC BNDPRM=' ',MEMSIZE=2048K,
//      STPLIB='ADA.V1R4M0.SEVGMOD1',MAXTIME=60,
//      VIO=VIO,SYSDA=SYSALLDA,SYSOUT='*'
//*****
//*                ERASE ADA.INFO DATASET
//*****
//      EXEC PGM=IEFBR14
//ADAINFO DD DSN=&USER..ADA.INFO,DISP=(MOD,DELETE),
//          SPACE=(1,1),UNIT=&SYSDA
//*****
//*                INVOKE THE BINDER
//*****
//STEP1  EXEC PGM=EVGINMB,PARM=' ( &BNDPRM',REGION=&MEMSIZE,
//          TIME=&MAXTIME,DYNAMNBR=65,COND=(4,LT)
//STEPLIB DD DSN=&STPLIB,DISP=SHR
//CONOUT  DD SYSOUT=&SYSOUT,DCB=(LRECL=120,BLKSIZE=120)
//ADAIN   DD DSN=&ADASRC,DISP=SHR,FREE=END,DCB=BUFNO=4
//ADAOUT  DD SYSOUT=&SYSOUT,
//          DCB=(RECFM=VB,LRECL=259,BLKSIZE=3120)
//ADALIB  DD DSN=&USER..ADA.LIBRARY,DISP=SHR
//ADAINFO DD DSN=&USER..ADA.INFO,DISP=(NEW,PASS,CATLG),
//          DCB=(RECFM=VB,LRECL=512,BLKSIZE=3120,DSORG=PS),
//          SPACE=(80,(10,50)),UNIT=&SYSDA
//ADAOBJ  DD DSN=&USER..OBJ,DISP=(MOD,CATLG,CATLG),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120,DSORG=PO,BUFNO=4),
//          SPACE=(80,(16000,16000,20)),UNIT=&SYSDA
//ADAMAP  DD DSN=&USER..ADAMAP,DISP=(MOD,CATLG,CATLG),
//          DCB=(RECFM=VB,LRECL=1023,BLKSIZE=3120,DSORG=PO,BUFNO=2),
//          SPACE=(132,(1000,2000,20)),UNIT=&SYSDA
//ADADMAP DD DSN=&USER..DEBUGMAP,DISP=(MOD,CATLG,CATLG),
//          DCB=(RECFM=VB,LRECL=1023,BLKSIZE=3120,DSORG=PO,BUFNO=2),
//          SPACE=(132,(1000,2000,20)),UNIT=&SYSDA
//ADALIST DD DSN=&USER..LISTING,DISP=(MOD,CATLG,CATLG),
//          DCB=(RECFM=VBA,LRECL=259,BLKSIZE=3120,DSORG=PO,BUFNO=2),
//          SPACE=(132,(500,2000,20)),UNIT=&SYSDA
//ADAUT1  DD SPACE=(4096,(50,100)),
//          DCB=(RECFM=FE,BLKSIZE=4096,DSORG=DA,BUFNO=3),
//          UNIT=&SYSDA
//ADAUT2  DD SPACE=(3120,(30,90)),
//          DCB=(RECFM=VB,LRECL=258,BLKSIZE=3120,DSORG=PS,BUFNO=3),
//          UNIT=&VIO
//ADAUT3  DD SPACE=(4096,(30,60)),
//          DCB=(RECFM=FB,BLKSIZE=4096,DSORG=DA,BUFNO=3),
//          UNIT=&SYSDA
//ADAUT4  DD SPACE=(4096,(30,60)),
//          DCB=(RECFM=FB,BLKSIZE=4096,DSORG=DA,BUFNO=3),
//          UNIT=&SYSDA
//ADAUT5  DD SPACE=(3120,(30,60)),
//          DCB=(RECFM=VB,LRECL=258,BLKSIZE=3120,DSORG=PS),
//          UNIT=&SYSDA

```



## Linking IBM Ada/370 Programs

This section describes the different ways you can invoke the linking step. Most often you will use the ADA command to invoke the linkage editor. However, you can also use JCL to link programs in the MVS batch environment. The linking of Ada programs that call non-Ada routines is discussed in the last part of this section.

## Using ADA Command to Link

To invoke the compile, bind, link and execute steps for a main program, compile it using the Run option of the ADA command. For example:

```
ADA source_file (RUN
```

will compile, bind, link and execute the program in *source\_file*.

### Under VM/CMS

The following example shows the correct procedure for breaking up the compiling and linking steps for an Ada program. It is preceded by a table explaining the names of files and compilation unit that it uses.

Table 3-2. VM/CMS Names Used in Example	
Name	Description
TEST ADA A	File containing source for the main program.
TEST	Compilation unit name.

```
ADA TEST ADA A
ADA TEST (BIND
LOAD TEST
GENMOD TEST
```

The first ADA command compiles the source file into the Ada library. The second ADA command invokes the Ada/370 binder to create the object file TEST TEXT A. The LOAD command loads the object files TEST TEXT into virtual storage and establishes the linkages. The GENMOD command uses the object file to create an executable load module with the name TEST MODULE A.

### Under MVS

The following example shows the correct procedure for breaking up the compiling and linking steps for an Ada program. It is preceded by a table explaining the names of files and compilation unit that it uses.

Table 3-3. MVS Names Used in Example	
Name in Example	Description
USER1.TEST.ADA	File containing source for the main program.
USER1.OBJ(TEST)	File containing object module of TEST.
USER1.LOAD(TEST)	File containing executable load module of TEST.
TEST	Compilation unit name.

```
ADA 'USER1.TEST.ADA'
ADA TEST (BIND NOGENERATE
```



## Linking Programs That Call Non-Ada Routines

The first ADA command compiles the source file into the Ada library. The second ADA command invokes the Ada/370 binder to create the object file USER1.OBJ(TEST). Then call the linkage editor:

```
LINK ('USER1.OBJ(TEST)') LOAD ('USER1.LOAD(TEST)')
```

The TEST load library member is now fully linked and ready to be executed.

**Note:** If you have installed ISPF/PDF under MVS, you can link edit your Ada/370 program by using the ISPF/PDF panels described in "ISPF/PDF Panels (MVS Only)" on page 2-28.

## Using Job Control Language to Link (MVS only)

This section describes how to invoke the linkage editor as a MVS batch job by using Job Control Language (JCL). The following example shows a job, called MYPROG, which invokes the linkage editor to create an executable load module in USER1.LOAD(HELLO). The input to the linkage editor is the Ada object code in USER1.OBJ(HELLO).

```
//MYPROG JOB, ' ',MSGCLASS=A,MSGLEVEL=(1,1),NOTIFY=USER1,
//      CLASS=A
//*-----
/*  PURPOSE : TO LINK AN ADA PROGRAM IN BATCH
/*-----
//LINKSTP EXEC PGM=IEWL,COND=(4,LT),REGION=0K,
//      PARM='RENT,SIZE=(2000K,500K),XREF'
//ADAOBJ DD DSN=USER1.OBJ,DISP=SHR
//SYSLIN DD *
//      INCLUDE ADAOBJ(HELLO) * THE ADA APPLICATION MODULE
//SYSLMOD DD DSN=USER1.LOAD(HELLO),DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSUT1 DD UNIT=VIO,SPACE=(1024,(120,120),,,ROUND),
//      DCB=BUFNO=1
```

## Linking Programs That Call Non-Ada Routines

Pragma INTERFACE enables Ada compilation units to call non-Ada routines. The following sections show you how to compile and link programs that take advantage of this feature.

When you link non-Ada routines and they are not inside the Ada library system, the only way you can connect non-Ada routines with the Ada routines that call them is to load them under VM/CMS or link-edit them under MVS. A call to a non-Ada routine results in the generation of an external reference. This external reference is unresolved following normal ADA binding. You must take special steps to resolve virtual address constants to non-Ada routines.

### Restrictions

1. Link names may overlap in C, FORTRAN, COBOL, and Ada runtime support. You must ensure that the links are resolved correctly among these runtime libraries.
2. Where available, an Ada program that calls a non-Ada subprogram must run in AMODE(31). The non-Ada subprogram must also run in AMODE(31).

You can use the transporter to move non-Ada code into a sublibrary together with the Ada code and package the whole application as a sublibrary. See "The ADATRANS Command" on page 5-29 on how to use the transporter.



**Under VM/CMS**

The following example shows the correct procedure for compiling and linking an Ada program with non-Ada routines. It is preceded by a table explaining the files that it uses.

Table 3-4. VM/CMS Files Used in Calling Non-Ada Routines	
File	Description
TEST ADA A	File containing source for the main program.
TEST	Compilation unit name.
TEST TEXT A	File containing the object code of the Ada routines.
ROUTINE TEXT A	File containing the object code of the non-Ada routines.

```
ADA TEST (MAIN NORUN
LOAD TEST ROUTINES
GENMOD TEST
```

The ADA command creates the object file TEST TEXT A. The LOAD command loads the object files TEST TEXT and ROUTINES TEXT into virtual storage and establishes the current linkages between them. The order in the LOAD is important. The Ada object must go first if it is to be the program's entry point. The GENMOD command uses the two object files to create a load module with the name TEST MODULE A.

You may need to precode the commands in this example with a GLOBAL TXTLIB command to resolve any missing external references from the LOAD command. The need for its use depends on how you load the non-Ada routines. For more information on the GLOBAL command, see the *Virtual Machine/System Product CMS Command and Macro Reference*.

**Under MVS**

The following table describes the data sets used in the example of linking an Ada program that calls non-Ada routines:

Table 3-5. MVS Data Sets Used in Calling Non-Ada Routines	
Data Set	Description
<i>qualifier</i> .TEST.ADA	Data set containing source for the main program, whose compilation unit name is Test.
<i>qualifier</i> .NONADA.OBJ	Data set containing the object code of the non-Ada routines.
<i>qualifier</i> .OBJ	Partitioned data set containing the object code of the Ada routines.
<i>qualifier</i> .LOAD	Partitioned data set containing the executable load modules; also called the "load library."

There are three methods for compiling and linking an Ada program with non-Ada routines. The first involves binding with NOGenerate and then linking the non-Ada language code manually with the linkage editor. The second involves placing the code in a partitioned data set (PDS) and associating it with the SYSLIB DD name so that the binder calls the linkage editor to resolve external references. The third uses JCL to call the linkage editor that creates executable load module from the object code stored in separate partitioned data sets. The following three examples show how to use these methods.



### **Using the Link Command:**

First compile the main program, using the NOGenerate option.

```
ADA TEST.ADA (MAIN NOGENERATE
```

This member contains one or more unresolved references to non-Ada code. The following call to the linkage editor resolves the unresolved external references associated with those calls.

```
LINK ('USER1.NONADA.OBJ','USER1.OBJ(TEST)') LOAD ('USER1.LOAD(TEST)')
```

The TEST load library member is now fully linked and ready to be executed.

### **Using a Partitioned Data Set:**

If you use a partitioned data set, follow these steps:

1. Place the object code of the non-Ada routine into a partitioned data set.
2. Issue the TSO ALLOC command for a DD name of SYSLIB, and then associate it with the PDS containing the non-Ada object code.
3. Bind the main program (or compile and bind) using the MAIN option. The LINK within ADA will refer to the SYSLIB allocation as it attempts to resolve references to the non-Ada routines.

```
ALLOC DD(SYSLIB) DA('USER1.NONADA.OBJ') SHR REU
```

The TEST load library member is now fully linked and ready to be executed.

For more information on LINK, see the LINK command in the IBM publication, *MVS/Extended Architecture TSO Extensions TSO Command Language Reference*.

### **Invoking the Linkage Editor with JCL:**

You can use JCL to invoke the linkage editor and create an executable load module from an Ada program that calls non-Ada routines. In the following example, the linkage editor takes the object code of the Ada program and the non-Ada routine that are in partitioned data sets called USER1.OBJ(TEST) and USER1.NONADA.OBJ(ASM1), respectively, and creates the executable load module in USER1.LOAD(TEST):

```
//MYPROG JOB,' ',MSGCLASS=A,MSGLEVEL=(1,1),NOTIFY=USER1,
//      CLASS=A
//-----
//*  PURPOSE : TO LINK AN ADA PROGRAM THAT CALL NON-ADA ROUTINE
//-----
//LINKSTP EXEC PGM=IEWL,COND=(4,LT),REGION=0K,
//      PARM='RENT,SIZE=(2000K,500K),XREF'
//ADAOBJ DD DSN=USER1.OBJ,DISP=SHR
//NADAOBJ DD DSN=USER1.NONADA.OBJ,DISP=SHR
//SYSLIN DD *
//      INCLUDE ADAOBJ(TEST) * THE ADA OBJECT
//      INCLUDE NADAOBJ(ASM1) * ASSEMBLER OBJECT
//SYSLMOD DD DSN=USER1.LOAD(TEST),DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSUT1 DD UNIT=VIO,SPACE=(1024,(120,120),,,ROUND),
//      DCB=BUFNO=1
```



## Binding/Linking a Collection of Exported Routines

The Bind option of the compiler must be used to create the object code for the exported Ada routines. There are two possible situations.

### When an Ada Routine Is the Main Program

No special options are required to bind a collection of Ada exported routines. Bind your program in the usual way.

### When a Non-Ada Routine Is the Main Program

In this case, the Export option must be used along with the Bind option. The syntax for binding this type of exported routines into an exported collection is described below:

`ADA compilation_unit_name ( Bind Export`

where:

`compilation_unit_name` Is a previously compiled unit and meets the requirements for a main program. Note that if you want to bind a collection that contains more than one routine or a single routine that violates the main program requirements, a dummy main program must be used. Refer to the *Ada/370 Programmer's Guide* for more information.

Export Is the option specifying that the output object deck is to be linked with a non-Ada main program.

The binder generates an object file to be linked with the non-Ada main program using the system linker or loader. The object file is generated using the normal conventions for the Bind option.

### Restriction

Two object modules output by the Bind step cannot be linked with the same non-Ada language main program. Duplicate CSECT IDs will result.



## APPENDIX C

### APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

package STANDARD is

type SHORT\_INTEGER is range -32768..32767;

type INTEGER is range -2147483648..2147483647;

type FLOAT is digits 6 range -7.23701E+75 .. 7.23701E+75;

type LONG\_FLOAT is digits 15 range

-7.23700557733225E+75 .. 7.23700557733225E+75;

type DURATION is delta 2\*\*(-14) range -86\_400.0..86\_400.0;

end STANDARD;



## Appendix F. Implementation-Dependent Features

The Ada language definition allows certain features to be different for different platforms. Because the standard defines which language features can be different, you should be able to anticipate and avoid portability problems in the code you write.

This section, called Appendix F as prescribed in the LRM, describes implementation-dependent characteristics of the IBM Ada/370 Release 4 licensed programs.

### Implementation-Defined Pragmas

The compiler supports the following implementation-dependent pragmas. These pragmas may have additional optional arguments that are for IBM use only. The ALLOCATION\_DATA and OS\_TASK pragmas only have an effect within MVS programs.

Form	Allowed Place	Effect
<code>pragma Active_Task;</code>	Within the task specification for a task type, or the specification of the task object for other kinds of tasks.	Prevents the compiler from applying passive tasking transformations to the task or task type. If both ACTIVE_TASK and PASSIVE_TASK pragmas are specified for the same task or task type, ACTIVE_TASK takes precedence. Refer to page 5-48 for full details.
<code>pragma Allocation_Data   (<i>access_type</i>,   <i>residence_mode</i>,   <i>allocation_duration</i>,   <i>subpool_number</i>,   <i>discrete_user_data</i>);</code>	Within the declarative part or package specification where the access type is declared. It does not have to immediately follow the access type declaration.	Pragma ALLOCATION_DATA associates MVS virtual storage attributes with an Ada access type. Objects of the access type are allocated in a parcel of storage whose attributes coincide with the attributes specified in the pragma. With this pragma, you can allocate storage for Ada objects in a specific subpool and in a specific XA residency mode (for instance, above the 16-megabyte line). Refer to page 2-13 for full details.
<code>pragma Comment   (<i>string_literal</i>);</code>	Any location within the source code of a compilation unit, except within the generic formal part of a generic unit.	Pragma COMMENT is used to insert header information into object code. The <i>string_literal</i> represents the characters to be embedded in the object code. The <i>string_literal</i> must fit on one line. Any number of comments can be entered into the object through pragma COMMENT.
<code>pragma Export   (<i>subprogram_name</i>,   "<i>OS_linkname</i>",   <i>language_name</i>);</code>	Within the same specification or declarative part which contains the subprogram declaration.	Pragma EXPORT allows Ada subprograms to be called from other languages. You can call a single Ada subprogram or a collection of Ada subprograms.
<code>pragma Images   (<i>enumeration_type</i>,   Deferred   Immediate);</code>	Within a package specification or declarative part, following the declaration of the specified enumeration type.	Pragma IMAGES controls when the generation of an image table occurs for an enumeration type. The compiler generates this image table for each enumeration type. Refer to page 6-15 for full details.



Form	Allowed Place	Effect
pragma Inline_Generic ( <i>instance_name</i>   <i>generic_name</i> );	Immediately following a generic specification or generic instantiation.	Causes the named instance, or all instances of the named generic unit, to be expanded inline. If the instantiation is overloaded in the scope in which it is declared, the pragma applies to all instances with that name. INLINE_GENERIC takes precedence over pragma SHARE_GENERIC and prevents the generic instance from being shared. If the compiler cannot expand the generic unit inline, it issues a warning, ignores the pragma, and uses deferred instantiation instead. Refer to page 6-8 for full details.
pragma Interface_Information ( <i>subprogram_name</i> , "linkname");	Immediately following a pragma Interface statement.	Pragma INTERFACE_INFORMATION is used in association with pragma Interface to provide access to any routine whose name can be specified by an Ada string literal. Refer to page 4-5 for full details.
pragma Interrupt ( <i>function_mapping</i> );	Preceding an <b>accept</b> statement in a task body.	Specifies that the following <b>accept</b> statement does not interact with other tasks, and so can be called directly from the context of an interrupted task. This saves the overhead of a rescheduling and rendezvous operation.  <b>Note:</b> The argument to this pragma is always the literal <i>function_mapping</i> . Case is not significant in the literal. Refer to page 5-45 for full details.
pragma No_Suppress ( <i>check_name</i> );	Same as pragma SUPPRESS.	Pragma NO_SUPPRESS prevents the compiler from suppressing checks within a particular scope. It is useful when a section of code that relies upon predefined checks executes correctly, but you need to suppress checks in the rest of the code for performance. You might use pragma SUPPRESS_ALL or the Check compiler option to suppress checks throughout an entire program, and then apply No_Suppress to only those compilation units that depend on the presence of error checks.
pragma Os_Task ( <i>priority</i> );	Anywhere within the specification of the task unit it designates, or within an Ada main subprogram.	Pragma OS_TASK specifically designates an MVS subtask to be executed as an asynchronous MVS subtask. The pragma takes advantage of the preemptive tasking feature of the MVS operating system. Refer to page 5-3 for full details.
pragma Passive_Task;	Within the task specification for a task type, or the specification of the task object for other kinds of tasks.	Requests that the compiler apply passive tasking transformations to the task or task type. The compiler issues a warning message if it cannot apply the transformation to a task with this pragma in its specification. The compiler only acts on this pragma when the compiler option Passive is specified. Refer to page 5-47 for full details.
pragma Preserve_Layout ( <i>record_type</i> );	Before any forcing occurrences of the record type <LRM 13.1>, in the same declarative part, package specification, or task specification as the record type declaration.	Specifies that the compiler should not reorder the components of the named record type. Refer to page 2-7 for full details.



Form	Allowed Place	Effect
pragma Share_Generic (package_name   subprogram_name);	Immediately following the declaration of a generic package or subprogram.	Specifies that all instances of the generic package or subprogram that are not expanded inline are generated in a way that allows identical copies of executable code to be shared. Refer to 6-14 for full details.
pragma Suppress_All;	Same as pragma SUPPRESS.	Invokes pragma SUPPRESS for all allowed condition names and for other run time checks such as elaboration checks. Refer to page 6-2 for full details.

## Predefined Pragmas

Supported pragmas are:

- ELABORATE
- INLINE (in units compiled with optimization)
- INTERFACE (to ASSEMBLER, FORTRAN, COBOL, and C.)
- LIST
- OPTIMIZE
- PACK
- PAGE
- PRIORITY
- SHARED
- SUPPRESS.

### Notes:

1. Pragma LIST suppresses listings prior to pragma LIST(ON), regardless of the user request.
2. The IBM products recognized by the pragma INTERFACE are ASSEMBLER, VS FORTRAN, C/370, and VS COBOL II.
3. When pragma PRIORITY is applied to a main program that uses the tasking facilities of pragma OS\_TASK, the priority argument is used to set the Ada tasking priority of the main program.
4. Although the compiler recognizes pragma INLINE, it only performs inline expansion when you compile with optimization enabled (through the Optimize compiler option).
5. Pragma MEMORY\_SIZE and pragma SYSTEM\_NAME are recognized, but the value cannot be changed.

The compiler displays an appropriate warning message when it finds an unrecognized or unsupported pragma.

## Implementation-Defined Attributes

There are some attributes that are for IBM use only. Do not code these attributes in any programs you write:

- OFFSET
- SUBPROGRAM\_VALUE
- ENTRY\_NUMBER
- TASK\_ID.



## Integer Type Attributes

### **Extended\_Image**(*Item*, *Width*, *Base*, *Based*, *Space\_If\_Positive*);

Returns the image associated with *Item* as defined in TEXT\_IO.INTEGER\_IO. The TEXT\_IO definition states that the value of *Item* is an integer literal with no underlines, no exponent, no leading zeros (except one leading zero if the item equals 0), and a minus sign if negative. All arguments except *Item* are optional.

### **Extended\_Value**(*Item*);

Returns the image associated with *Item* as defined in TEXT\_IO.INTEGER\_IO. The TEXT\_IO definition states that, given a string, it reads an integer value from the beginning of the string. The value returned corresponds to the sequence input.

### **Extended\_Width**(*Base*, *Based*, *Space\_If\_Positive*);

Returns the width for a specified subtype. All arguments are optional.

## Enumeration Type Attributes

### **Extended\_Image**(*Item*, *Width*, *Uppercase*);

Returns the image associated with *Item* as defined in TEXT\_IO.ENUMERATION\_IO. The TEXT\_IO definition states that, given an enumeration literal, it returns the value of the enumeration literal, either an identifier or a character literal. The character case parameter is ignored for character literals. All arguments except *Item* are optional.

### **Extended\_Value**(*Item*);

Returns the image associated with *Item* as defined in TEXT\_IO.ENUMERATION\_IO. The TEXT\_IO definition states that it reads an enumeration value from the beginning of the string and returns the value of the enumeration literal that corresponds to the sequence input.

### **Extended\_Width**;

Returns the width for a specified subtype.

## Floating-Point Type Attributes

### **Extended\_Image**(*Item*, *Fore*, *Aft*, *Exp*, *Base*, *Based*);

Returns the image associated with *Item* as defined in TEXT\_IO.FLOAT\_IO. The TEXT\_IO definition states that it returns the value of the parameter *Item* as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of *Item*. If *Exp* is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of *Item*, or is 0 if the value of *Item* has no integer part. All arguments except *Item* are optional.

### **Extended\_Value**(*Item*);

Returns the image associated with *Item* as defined in TEXT\_IO.FLOAT\_IO. The TEXT\_IO definition states that it skips any leading zeros, then reads a plus or minus sign if present, then reads the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input.

### **Extended\_Digits**(*Base*);

Returns the number of digits using *Base* in the mantissa of model numbers of the specified subtype. *Base* is optional.



## Fixed-Point Attributes

### **Extended\_Image(*Item*, *Fore*, *Aft*, *Exp*, *Base*, *Based*);**

Returns the image associated with *Item* as defined in TEXT\_IO.FIXED\_IO. The TEXT\_IO definition states that it returns the value of the parameter *Item* as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of *Item*. If *Exp* is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of *Item*, or is 0 if the value of *Item* has no integer part. All arguments except *Item* are optional.

### **Extended\_Value(*Item*);**

Returns the image associated with *Item* as defined in TEXT\_IO.FLOAT\_IO. The TEXT\_IO definition states that it skips any leading zeros, then reads a plus or minus sign if present, then reads the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input.

### **Extended\_Fore(*Base*, *Based*);**

Returns the minimum number of characters required for the integer part of the based representation specified. All arguments are optional.

### **Extended\_Aft(*Base*, *Based*);**

Returns the minimum number of characters required for the fractional part of the based representation specified. All arguments are optional.

---

## Package SYSTEM

The current specification of package SYSTEM includes the following:

```
=====
-- CUSTOMIZABLE VALUES
=====
```

```
type Name      is (MC68000, ANUYK44, IBM370);
```

```
System_Name    : constant name := IBM370;
```

```
Memory_Size    : constant := (2 ** 31) - 1; --Available memor , in storage units
Tick           : constant := 1.0 / (10 ** 6);
```

```
=====
-- NON-CUSTOMIZABLE, IMPLEMENTATION-DEPENDENT VALUES
=====
```

```
Storage_Unit   : constant := 8;
Min_Int        : constant := -(2 ** 31);
Max_Int        : constant := (2 ** 31) - 1;
Max_Digits     : constant := 15;
Max_Mantissa   : constant := 31;
Fine_Delta     : constant := 1.0 / (2 ** Max_Mantissa);
```

```
subtype Priority is Integer range 0..255;
```

```
=====
-- ADDRESS TYPE SUPPORT
=====
```

```
type Memory is private;
```



```

type Address is access Memory;
--
-- Ensures compatibility between addresses and access types.
-- Also provides implicit NULL initial value.

Null_Address: constant Address := null;
--
-- Initial value for any Address object
type Address_Value is range -(2**31)..(2**31)-1;
--
-- A numeric representation of logical addresses for use in address clauses

Hex_80000000 : constant Address_Value := - 16#800000000#;
Hex_90000000 : constant Address_Value := - 16#700000000#;
Hex_A0000000 : constant Address_Value := - 16#600000000#;
Hex_B0000000 : constant Address_Value := - 16#500000000#;
Hex_C0000000 : constant Address_Value := - 16#400000000#;
Hex_D0000000 : constant Address_Value := - 16#300000000#;
Hex_E0000000 : constant Address_Value := - 16#200000000#;
Hex_F0000000 : constant Address_Value := - 16#100000000#;
--
-- Define numeric offsets to aid in Address calculations
-- Example:
--   for Hardware use at Location (Hex_F0000000 + 16#2345678#);

function Location is new Unchecked_Conversion (Address_Value, Address);
--
-- May be used in address clauses:
--
--   Object: Some_Type;
--   for Object use at Location (16#4000#);

function Label (Name: String) return Address;
--
-- The LABEL function allows a link name to be specified as address
-- for an imported object in an address clause:
--
--   Object: Some_Type;
--   for Object use at Label("SYSTEM_DATA_AREA");
--
-- System.Label returns Null_Address for non-literal parameters.

Max_Object_Size   : CONSTANT := Max_Int;
Max_Record_Count  : CONSTANT := Max_Int;
Max_Text_Io_Count : CONSTANT := Max_Int-1;
Max_Text_Io_Field : CONSTANT := 1000;

```

---

## Representation Clauses

This implementation supports address, length, enumeration, and record representation clauses with the following exceptions:

Address clauses for the name of an Ada subprogram, package, or task unit <LRM 13.5(b)> are not supported. You can examine the ADDRESS attribute of subprograms written in other languages and designated by pragma INTERFACE.



Address clauses for the name of a single entry <LRM 13.5(c)>are supported under MVS only. For information about creating interrupt entries for MVS, see "The Ada/370 Interrupt Model (MVS Only)" on page 5-41.

Enumeration clauses are not supported for Boolean representation clauses.

The size in bits of representation specified records is rounded up to the next highest multiple of 8, meaning that the object of a representation specified record with 25 bits will actually occupy 32 bits, and, if the record is used as a component of another representation specified record, 32 bits must be reserved for it.

Unsupported clauses are rejected at compile time.

For a more detailed explanation, see "Representation Clauses" on page 2-5.

## Interpretation of Expressions in Address Clauses

Expressions that appear in address clauses are interpreted as virtual memory addresses.

## Implementation-Dependent Constraints

For capacities shown as unlimited in the following table, no upper limit has been built into the compiler. The capacities are still limited by the amount of storage on your system. A program can exceed the compiler's capacity limits if it makes heavy use of more than one capability that, by itself, is considered unlimited.

Capability	Compiler Limit
Maximum nesting of subprograms	unlimited
Maximum length of an input line	200 characters
Library units in a library	400_000
Compilation units in a program	400_000
Ada source statements in a program	unlimited
ELABORATE pragmas in a program	unlimited
Width of a source line	200
Length of an identifier	200
Library units in a single context clause	unlimited
Library units used in <b>with</b> clauses by a compilation unit	unlimited
External names	unlimited
Ada source statements in a single compilation unit	32_767
Identifiers (including those in units included by <b>with</b> clauses)	unlimited
Declarations (total) in a compilation unit	unlimited
Subtype declarations of a single type	unlimited
Literals in a compilation unit	unlimited
Depth of nesting of program units	unlimited
Depth of nesting of blocks	unlimited
Depth of nesting of <b>case</b> statements	unlimited
Depth of nesting of <b>loop</b> statements	unlimited



Capability	Compiler Limit
Depth of nesting of <b>if</b> statements	unlimited
<b>elsif</b> alternatives	unlimited
Exception declarations in a frame	unlimited
Exception handlers in a frame	unlimited
Declarations in a declarative part	unlimited
Frames an exception can propagate through	unlimited
Values in subtype <b>SYSTEM.PRIORITY</b>	256
Simultaneously active tasks in a program	unlimited
<b>accept</b> statements in a task	unlimited
<b>entry</b> declarations in a task	unlimited
Formal parameters in an <b>entry</b> declaration	4016
Formal parameters in an <b>accept</b> statement	4016
Delay statements in a task	unlimited
Alternatives in a <b>select</b> statement	unlimited
Formal parameters to a subprogram	4016
Levels in a call chain	unlimited
Visible declarations in a package	unlimited
Package declarations	unlimited
Declarations in a block	unlimited
Enumeration literals in a type	Natural'First .. Natural'Last
Dimensions in an array	unlimited
Total elements in an array	Integer'First .. Integer'Last
Components in a record type	unlimited
Discriminants in a record type	unlimited
Variant parts in a record type	unlimited
Size of any object in bytes	System.Max_Int / 8
Characters in a value of type <b>STRING</b>	(2**31)-1
Operators in an expression	unlimited
Function calls in an expression	unlimited
Primaries in an expression	unlimited
Depth of parentheses nesting	unlimited

## Implementation-Generated Names

There are no implementation-generated names that denote implementation-dependent components. Names generated by the compiler do not interfere with programmer-defined names.



---

## Unchecked Conversion Restrictions

Unchecked conversions are allowed between types (or subtypes) T1 and T2 provided that:

- They are constrained.
- They are not private.

If the sizes of objects of the source and target types are static and equal, the compiler performs a bitwise copy of data from one object to the other. Refer to Table 2-1 on page 2-3 to see how the compiler computes the sizes for different types and objects.

When the sizes of the objects differ:

- If the source object is larger than the target object, the most significant bits are truncated in the conversion.
- If the source object is smaller than the target object, the source is copied to the least significant bits of the target and the remaining bits have unpredictable values.

The compiler issues a warning when you instantiate UNCHECKED\_CONVERSION for types whose objects have different or nonstatic sizes. Performing conversions for objects of different or nonstatic sizes may be less efficient.

---

## Implementation-Dependent Characteristics of the I/O Packages

This is a summary of the I/O specifics for Ada/370. See "Performing Input/Output Operations" on page 3-1 for a detailed explanation.

- SEQUENTIAL\_IO, DIRECT\_IO, and TEXT\_IO are supported.
- LOW\_LEVEL\_IO is not supported.
- Unconstrained array types and unconstrained types with discriminants cannot be used to instantiate DIRECT\_IO.
- VM/CMS and MVS file names follow the conventions and restrictions of their respective operating systems. These are explained in detail in "NAME Parameters" on page 3-2.
- In TEXT\_IO, the type Field is defined as follows:  
    subtype Field is integer range 0..1000;
- In TEXT\_IO, the type Count is defined as follows:  
    type Count is range 0..2\_147\_483\_645;
- The conditions under which I/O exceptions are raised are discussed in "Anticipating I/O Exceptions" on page 3-27.



## Form Parameters

Table F-1. Keywords for VM/CMS FORM Strings

Keyword	Parameter	CREATE Default	OPEN Default
RECFM	F or V	F for constrained SEQUENTIAL_IO, and DIRECT_IO packages. V for unconstrained SEQUENTIAL_IO, TEXT_IO, and E_TEXT_IO packages.	Do not need to specify when opening a file. USE_ERROR is raised if values do not match those specified when the file was created.
LRECL	integer	Length of type or subtype for SEQUENTIAL_IO, and DIRECT_IO packages. LRECL is optional for TEXT_IO and E_TEXT_IO packages.	Do not need to specify when opening a file. USE_ERROR is raised if values do not match those specified when the file was created.

Table F-2 (Page 1 of 2). Keywords for MVS FORM Strings

Keyword	Parameter	Description
ASYNCHRONOUS	none	Support asynchronous I/O for this data set. This is the default for DIRECT_IO and SEQUENTIAL_IO. TEXT_IO, and E_TEXT_IO operations are always asynchronous, so neither SYNCHRONOUS nor ASYNCHRONOUS can be specified in the form string. SYNCHRONOUS and ASYNCHRONOUS are mutually exclusive.
BLKSIZE	integer	Data-set block size in bytes. This keyword is not valid on an OPEN subprogram; if specified, it will be ignored.
BUFFERS	integer	The number of channel programs and buffers used for I/O operations on the specified data set.
DIRECTORY	integer	The number of 256-byte directory records to allocate to the directory of a PDS. This keyword is not valid on an OPEN subprogram; if specified, it will be ignored.
EXCLUSIVE	none	Ensure your program has exclusive access to the file.
LRECL	integer	Set the record length for a data set created by the CREATE subprogram in the SEQUENTIAL_IO, DIRECT_IO, TEXT_IO, and E_TEXT_IO packages. This keyword is only valid on a CREATE subprogram.



Table F-2 (Page 2 of 2). Keywords for MVS FORM Strings

Keyword	Parameter	Description
MEMBER	data-set member name	Specify which member of a PDS to open to I/O. In this case, the name string must begin with "=" to designate a preallocated data set.
RECFM	F, FA, V, VB, FB, FBA, VA, VBA	Define the record format for a data set created by the CREATE subprogram in the SEQUENTIAL_IO, DIRECT_IO, TEXT_IO, and E_TEXT_IO packages. This keyword is only valid on a CREATE subprogram.
PRIMARY	integer	Data-set primary extent size in blocks. This keyword is not valid on an OPEN subprogram; if specified, it will be ignored.
SECONDARY	integer	Data-set secondary extent size in blocks. This keyword is not valid on an OPEN subprogram; if specified, it will be ignored.
SYNCHRONOUS	none	Do not perform asynchronous I/O for this data set.
UNIT	name or number	The disk unit device name or virtual input/output (VIO) unit name. The name to use may differ between sites.
VOLSER	name or number	The volume serial number of a DASD or tape device.

## Representation Attributes for Real Types

The actual representation of the floating-point numbers in Ada/370 is mapped onto the System/370 single-precision floating-point representation:

- The attribute MACHINE\_ROUNDS for a real type indicates whether floating-point results are exact or rounded. On the System/370 hardware, floating-point results are truncated, not rounded, when the result is stored into the result register, so the MACHINE\_ROUNDS attribute is always FALSE.
- The attribute MACHINE\_RADIX specifies the radix used for the machine representation of real numbers. System/370 computers treat the mantissa as hexadecimal digits, so the MACHINE\_RADIX attribute is always 16.
- The attribute MACHINE\_MANTISSA for real types indicates the number of machine radix digits used for representing the mantissa. For single-precision floating-point, the 24 bits of the System/370 fraction holds 6 hexadecimal digits, so the MACHINE\_MANTISSA attribute is 6.

For double-precision floating-point, the 56 bits of the System/370 fraction holds 15 hexadecimal digits, so the MACHINE\_MANTISSA attribute is 15.

- The attributes MACHINE\_EMIN and MACHINE\_EMAX for real types correspond respectively to the smallest (most negative) and largest value of the exponent. For each Ada/370 floating-point type, MACHINE\_EMIN is -64 and MACHINE\_EMAX is 63.
- The attribute MACHINE\_OVERFLOW indicates whether a NUMERIC\_ERROR exception is raised when an overflow condition is encountered. Not all cases are detected, and the NUMERIC\_ERROR exception is sometimes not raised, so the MACHINE\_OVERFLOW attribute is always FALSE.



---

## Attributes of Predefined Numeric Types

These attributes are part of the predefined package STANDARD. For more discussion of the way the compiler determines the 'SIZE attribute for data types and objects, see Table 2-1 on page 2-3.

### SHORT\_INTEGER

'First = -32\_768  
'Last = 32\_767  
'Size = 16

### INTEGER

'First = -2\_147\_483\_648  
'Last = 2\_147\_483\_647  
'Size = 32

### FLOAT

'Digits = 6  
'Emax = 84  
'Epsilon = 9.53674E-07  
'Large = 1.93428E+25  
'Machine\_Emax = 63  
'Machine\_Emin = -64  
'Machine\_Mantissa = 6  
'Machine\_Overflows = FALSE  
'Machine\_Radix = 16  
'Machine\_Rounds = FALSE  
'Mantissa = 21  
'Safe\_Emax = 252  
'Safe\_Large = 7.23700E+75  
'Safe\_Small = 6.90893E-77  
'Size = 32  
'Small = 2.58494E-26

### LONG\_FLOAT

'Digits = 15  
'Emax = 204  
'Epsilon = 8.88178419700125E-16  
'Large = 2.57110087081438E+61  
'Machine\_Emax = 63  
'Machine\_Emin = -64  
'Machine\_Mantissa = 14  
'Machine\_Overflows = FALSE  
'Machine\_Radix = 16  
'Machine\_Rounds = FALSE  
'Mantissa = 51  
'Safe\_Emax = 252  
'Safe\_Large = 7.23700557733226E+75  
'Safe\_Small = 6.90893484407556E-77  
'Size = 64  
'Small = 1.94469227433161E-62

### DURATION

'Delta = 2\*\*(-14)  
'First = -86\_400.0



'Last	= 86_400.0
'Machine_Overflows	= FALSE
'Machine_Rounds	= FALSE

---

## Parameters to a Main Subprogram

The subprogram specified as the main subprogram during the bind step of compilation must not accept any Ada parameters.

For run time parameters specified on the command line, you can retrieve them by using the `COMMAND_LINE` package described in "Package `COMMAND_LINE`" on page 2-17.